

TURING 图灵程序设计丛书

Big Nerd Ranch

■ 通过4个实战项目
全面掌握Web开发

Web开发权威指南

[美] Chris Aquino, Todd Gandee 著
奇舞团 译

Front-End Web Development
The Big Nerd Ranch Guide

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

译者简介

本书译者均来自奇虎360前端团队“奇舞团”

孟之杰

技术翻译爱好者，具有Geek精神，
喜欢折腾各种有意思的东西。

黄小璐

毕业于华中科技大学计算机学院，参与过的开源项目包括ThinkJS（基于Node的Web框架）和STC（高性能前端工作流系统），参与翻译了《高性能HTML5》《移动Web手册》《大型JavaScript应用最佳实践指南》。

钟　恒

《响应式Web设计：HTML5和CSS3实战（第2版）》译者，曾在QCon、SDCC、SFDC等大会上发表演讲。

卢士杰

燕尾服2.0与ThinkJS3开源项目贡献者，
360前端静态资源库网站开发与维护者。

孟芊冶

ACMer，毕业于哈尔滨工程大学软件学院，
爱好翻译。

文　蔺

毕业于国内某知名新闻学院，技术翻译爱好者，
前端公众号“奇舞周刊”编辑。

前端公众号“奇舞周刊”

TURING 图灵程序设计丛书



Web开发权威指南



[美] Chris Aquino, Todd Gandee 著

奇舞团 译

Front-End Web Development

The Big Nerd Ranch Guide

人民邮电出版社

北京

图书在版编目（C I P）数据

Web开发权威指南 / (美) 克里斯·阿基诺 (Chris Aquino) , (美) 托德·甘迪 (Todd Gandee) 著 ; 奇舞团译. -- 北京 : 人民邮电出版社, 2017.9
（图灵程序设计丛书）
ISBN 978-7-115-46616-7

Ⅰ. ①W… Ⅱ. ①克… ②托… ③奇… Ⅲ. ①网页制作工具－程序设计－指南 Ⅳ. ①TP393.092.2-62

中国版本图书馆CIP数据核字(2017)第193670号

内 容 提 要

本书在知名培训机构 Big Nerd Ranch 培训教材的基础上编写而成，囊括了 JavaScript、HTML5、CSS3 等现代前端开发人员急需的技术关键点，包括响应式 UI、访问远程 Web 服务、用 Ember.js 构建应用，等等。此外，还会介绍如何使用前沿开发工具来调试和测试代码，并且充分利用 Node.js 和各种开源的 npm 模块的强大功能来进行开发。

全书分四部分，每部分独立完成一个项目，由浅入深、循序渐进，在构建一系列应用的过程中，介绍 Web 开发的核心概念和 API。

无论是否拥有 Web 开发经验，抑或拥有其他平台的开发背景，只要对当今流行的工具和开发实践充满兴趣，这本书都能让你受益匪浅。

◆ 著　　　[美] Chris Aquino, Todd Gandee
译　　　奇舞团
责任编辑　朱　巍
执行编辑　夏静文
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：26.75
字数：632千字　　　　2017年 9 月第 1 版
印数：1 – 4 000册　　　　2017年 9 月北京第 1 次印刷

著作权合同登记号　图字：01-2016-7262号

定价：99.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

献　　词

感谢父母为我们购置了一台电脑；感谢Dave和Glenn让你们的弟弟我完全霸占了那台电脑；感谢Angela让我拥有电脑世界之外的美好生活。

—— C.A.

谢谢父母给我足够的空间，让我找到自己的方向；感谢我的妻子，谢谢你对我这个书呆子的爱。

—— T.G.

前　　言

学习前端 Web 开发

进行前端Web开发可能需要转换一下观念，因为它跟其他平台上的开发有很大不同。在学习过程中，你需要时刻牢记以下几点。

浏览器是一个平台

你也许在iOS或者Android上进行过原生开发，或者用Ruby、PHP写过服务器端代码，抑或在OS X或者Windows上构建过桌面应用。但作为前端开发者，你的代码则需要面向浏览器——一个几乎存在于全世界所有手机、平板电脑和个人计算机中的平台。

前端开发横跨一个范围

这个范围的一端是网页的外观和风格（圆角、阴影、颜色、字体、空白等），另一端则是控制网页复杂行为的逻辑（浏览交互式相册时滑动图片、校验表单数据、通过聊天网络发送消息等）。你需要通晓这个范围内的每种核心技术，还经常需要搭配使用多种技术来实现优秀的Web应用。

Web技术是开放的

没有哪家公司能够控制浏览器的工作方式。也就是说，前端开发者并不会每年得到一个SDK版本，而且这个版本里还包含了未来一年中可能要处理的所有改变。原生平台就像结了冰的池塘，任你舒适地滑过；而Web就像河流，蜿蜒曲折，水流湍急，某些地方还会有礁石——不过这正是它的魅力所在。Web是进化最快的平台，适应变化才是前端开发者的生存之道。

本书的目标是教会你如何在浏览器上进行开发。在本书的指导下，你将会经历一系列项目的开发，而每个项目都需要搭配使用前端范围内的不同技术。因为前端可用的工具、库以及框架不计其数，所以本书主要使用最重要也最便于移植的模式和技术。

目标读者

这并不是一本介绍编程的书。本书假定你已经具备编写代码的基础知识，并熟悉基本的类型、函数和对象。

话虽如此，但本书不要求你了解JavaScript。根据需要，本书会在具体语境中介绍JavaScript的相关概念。

本书的组织结构

本书会指导你实现四个不同的Web应用。每个应用对应书中的一个部分，每个部分的每一章会向当前正在构建的应用添加新功能。

构建这4个应用的过程横跨整个前端范围。

Ottergram	第一个项目是一个基于Web的图片浏览应用。通过构建Ottergram，能教会你通过使用HTML、CSS以及JavaScript进行浏览器编程的基础知识。你将手动构建用户界面（User Interface，UI），并且掌握浏览器加载和渲染内容的方式
CoffeeRun	CoffeeRun的一部分是咖啡订购表单，另一部分是清单。构建本应用涉及一系列JavaScript技术，包括编写模块代码、使用闭包，以及使用Ajax与远程服务器通信。你的关注点会从之前的手动创建UI转移到通过编程创建和操作UI
Chattrbox	Chattrbox的内容最少，但也最特别。你将用JavaScript创建一个聊天系统，用Node.js编写一个聊天服务器和一个基于浏览器的聊天客户端
Tracker	最后一个项目将使用Ember.js，它是前端开发最强大的框架之一。你将会创建一个应用，用来收录人们见过的奇异、神秘的珍稀生物。在开发过程中，你会学习支撑Ember.js框架的丰富的生态系统

在开发这些应用的过程中，你将会学习使用很多工具，包括：

- Atom文本编辑器和一些方便代码编写的插件
- 文档资源，比如Mozilla Developer Network（MDN）
- 命令行，使用OS X终端应用或者Windows命令行
- browser-sync
- Google Chrome开发者工具
- normalize.css
- Bootstrap
- jQuery以及库函数，比如crypto-js和moment
- Node.js、Node包管理工具（Node package manager，npm）以及nodemon
- WebSockets和wscat模块
- Babel、Babelify、Browserify以及Watchify
- Ember.js和插件，比如Ember CLI、Ember Inspector、Ember CLI Mirage以及Handlebars
- Bower
- Homebrew
- Watchman

如何使用本书

不同于参考手册，本书的目标是带你入门，让你学到参考手册上所教的大部分知识。本书基于Big Nerd Ranch的5天课程，因此会从入门知识开始。每一章都基于前面的知识，所以跳跃式阅读可能会影响学习效果。

在我们的课堂上，学生会学习本书的内容，但是他们还能享受一个良好的环境——浓厚的学习氛围、好吃的食物、相互鼓励的同学，还有答疑解惑的辅导老师。

作为读者，你也会希望有相似的环境。所以，睡个好觉，找一个安静的地方开始学习吧。做下面这些事情也会有所帮助。

- 与朋友或同事组成一个阅读小组。
- 安排一段时间集中学习一些章节。
- 参与本书论坛forums.bignerdranch.com上的讨论，在那儿你能探讨书中内容，发现勘误并找到解决方案。
- 找个了解前端开发的人帮你。

挑战

几乎每一章的结尾部分都会有至少一个挑战。这些挑战能帮你复习所学知识，并将这一章中做的项目再推进一步。建议你尽量完成这些挑战，这样可以巩固所学的知识，并且从**学习**JavaScript开发变为自己**从事**JavaScript开发。

这些挑战的难度分为3个级别。

- 初级挑战一般要求你做一些与该章内容相似的事情。这些挑战旨在强化所学内容，迫使你在不看代码的情况下写出相似的代码。熟能生巧就是这个道理。
- 中级挑战要求你挖掘更深，思考更多。有时候需要使用你从未见过的函数、事件、标记以及样式，但是任务还是与该章的任务类似。
- 高级挑战很难，可能需要花好几个小时才能完成。它们要求你理解该章介绍的概念，做一些质量层面的思考，并自己解决问题。解决这些问题可以为你在现实世界中进行JavaScript开发做好准备。

进行每一章的挑战之前都复制一份代码，否则你做的改动可能无法兼容后续的练习。

如果你感到困惑，记得访问forums.bignerdranch.com寻求帮助。

延展阅读

每章结尾的“延展阅读”对该章话题做了更深入的解释，或者提供了附加信息。这部分信息并不一定重要，但是希望你觉得它们有趣或者有用。

电子书

如需购买本书的电子版，请扫描如下二维码。

致　　谢

作为作者，我们完全可以把本书中的文字和图表都归功于我们自己。（棒呀，我们！）但实际上，如果没有投稿人、协作者和导师们的共同努力，我们可能到现在都无从下笔。

- Aaron Hillegass相信我俩能写出配得上Big Nerd Ranch之名的作品。你给予了我们极大的信心和支持，谢谢！
- Matt Mathias在本书的创作过程中给予了我们指导，对最后关键部分的建议尤为重要。你督促我们把时间花在写作上，而不是花在看猫咪视频或者《唐顿庄园》的重播上。
- Brandy Porter一直照料我俩并给我们烹饪美食。你在幕后做了各种工作，为本书的顺利出版打点好了一系列事情。谢谢你。
- Jonathan Martin是我们的导师和语言专家。谢谢你充满热情地教我们尚未完结的课程内容，本书就是基于这些内容写成的。在多次审校中，你还提供了很有深度的意见和建议。
- 我们的校对员、技术审稿人以及实验对象：Mike Zornek、Jeremy Sherman、Josh Justice、Jason Reece、Garry Smith、Andrew Jones、Stephen Christopher以及Bill Phillips。谢谢你们的志愿工作。
- Elizabeth Holaday是一位极具耐心且令人安心的编辑。谢谢你打破我们狭隘的思维，提出有理有据的观点，并提醒我们始终要考虑到读者。
- Ellie Volckhausen，谢谢你设计了封面。
- Simone Payment是我们的校对员，谢谢你让本书内容连贯一致。
- 来自IntelligentEnglish.com的Chris Loper设计并制作了本书的印刷版和电子版。另外，他的DocBook工具链非常好用。

最后，感谢无数接受了长达一周的培训的学生们。如果没有你们的好奇心和提问，培训就毫无意义。这本书正是得益于你们在这么多周的培训中的见解和灵感。我们希望那些水獭让培训轻松了一些。

目　　录

第四部分 应用架构

第一部分

浏览器编程基础

第 1 章

配置开发环境

前端开发领域有着数不清的工具和资源，而且还有更多工具在被源源不断地制造出来。无论开发者的水平如何，选择最佳工具都极具挑战性。本书的项目将会带你使用一些本书作者非常喜爱的工具。

首先你需要三个基本工具：浏览器、文本编辑器，以及好用的前端开发技术参考文档。此外还有一些能够提升开发体验的附加选项，当然它们并非必需品。

为达到最佳效果，建议你和本书作者使用相同的软件。本章会引导你安装并配置Google Chrome浏览器、Atom文本编辑器、Node.js以及一些插件。另外还会介绍一些优秀文档，并针对Mac和Windows**命令行**进行一次突击学习。在下一章开始第一个项目时，这些资源将会派上用场。

1.1 安装 Google Chrome

默认情况下，你的电脑应该已安装过浏览器，但前端开发中最好用的浏览器还是Google Chrome。若尚未安装最新版Chrome，可以通过www.google.com/chrome/browser/desktop获取（如图1-1所示）。

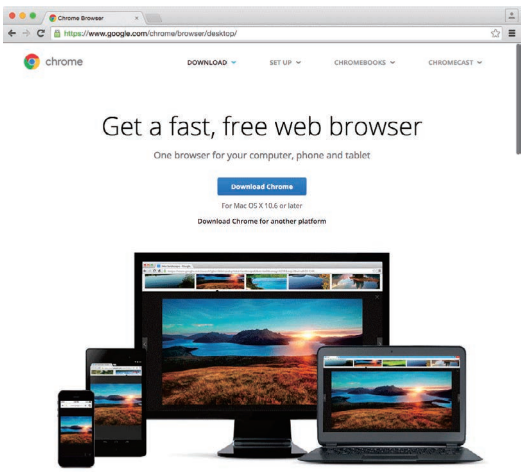


图1-1　下载Google Chrome

1.2　安装并配置 Atom

在众多文本编辑器中，GitHub出品的Atom是前端开发的最佳选择之一。它可以进行大量配置，也有很多辅助编码的插件。另外，它可以免费下载使用。

可以通过atom.io下载Mac版或Windows版的Atom（如图1-2所示）。

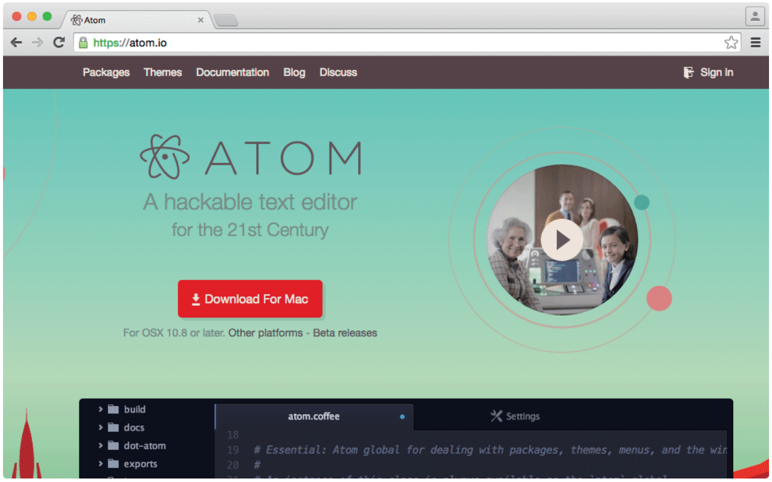


图1-2　下载Atom

遵循安装指引即可。安装成功后，还需要安装一些可能用到的插件。

Atom插件

对于一款文本编辑器来说，最需要的无非是文档查询、自动补全、代码格式化以及代码提示（接下来会细说）功能。Atom默认提供了一部分功能，但安装一些插件后效果会更好。

打开Atom的Settings面板。在Mac上，选择Atom → Pre ferences...，或者使用Command + ,（Command加逗号）快捷键。在Windows上，点击File → Settings或者按Control + ,快捷键。

左侧就是Settings面板，点击+ Install（如图1-3所示）。

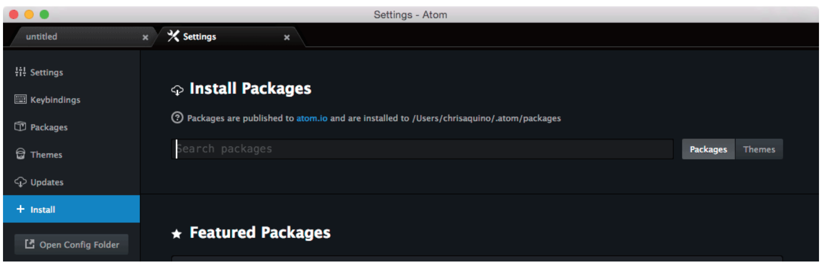


图1-3　Atom Install Packages面板

可以根据名称搜索插件，先搜索“emmet”试试看。

书写大量HTML是很枯燥的，而且容易出错。通过emmet插件（如图1-4所示），可以用一些速记符生成符合语法规则的HTML。点击Install按钮开始安装emmet。

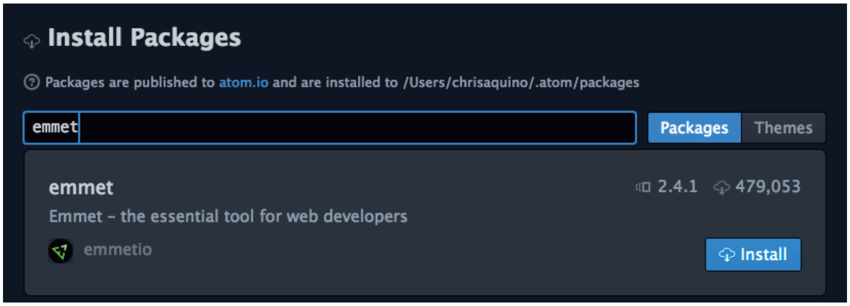


图1-4　安装emmet

然后再搜索“atom-beautify”。该插件（如图1-5所示）可辅助缩进，提高代码可读性。同样还是点击Install进行安装。

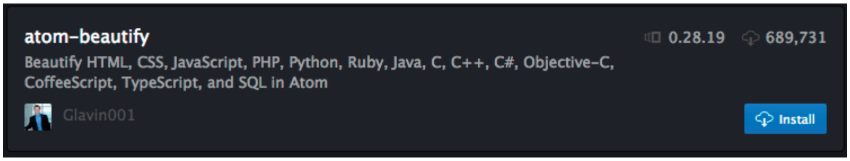


图1-5　安装atom-beautify

接着搜索并安装autocomplete-paths（如图1-6所示）。在代码中，经常需要引用其他文件或目录。该插件能够在输入文件名时提供自动补全功能。



图1-6　安装autocomplete-paths

接下来安装api-docs（如图1-7所示），这样就可以通过键盘查阅文档。文档会在编辑器单独的一个标签面板中显示。

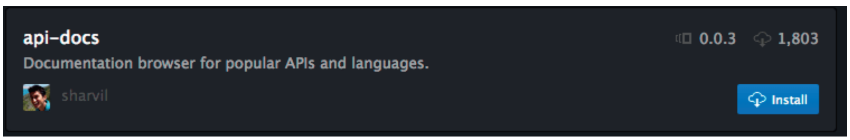


图1-7　安装api-docs

下面安装linter（如图1-8所示）。linter是一个用来检查代码语法和风格的程序。请确认你安装的插件就叫作linter。这是一个基础linter，与特定语言插件搭配使用效果更佳；有了它，才能使用后续其他linter插件。

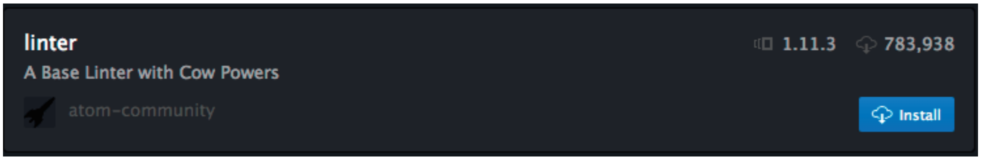


图1-8　安装linter

要使用linter检查CSS、HTML和JavaScript代码，还需安装另外三个插件。首先安装linter-csslint（如图1-9所示），该插件在保证CSS代码语法正确的同时，还能提供编写高性能CSS的建议。

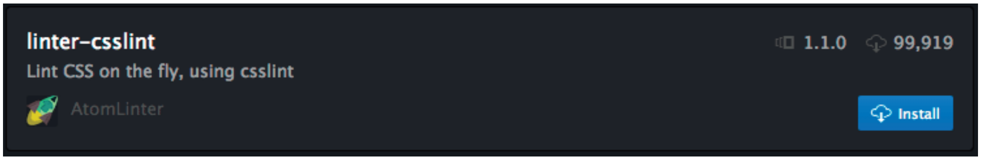


图1-9　安装linter-csslint

然后是linter-htmlhint（如图1-10所示），它确保HTML代码保持良好的格式。当HTML标签匹配错误时，编辑器会显示警告信息。



图1-10　安装linter-htmlhint

最后一个linter插件是linter-eslint（如图1-11所示）。它能检查JavaScript代码的语法，还能通过配置检查代码的风格和格式（如每行缩进几个空格，注释前后空几行等）。

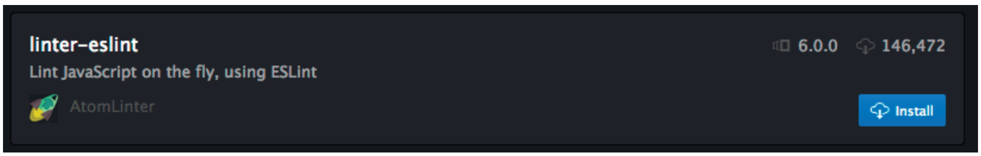


图1-11　安装linter-eslint

现在Chrome和Atom都安装好了，但还需要做一些完善编码环境的工作：访问参考文档、学习命令行基础知识、安装最后两个工具。

1.3　文档和参考资料

前端开发与针对iOS和Android等平台的开发有所不同。抛开显而易见的差异不说，除了技术规范之外，前端技术是没有官方开发者文档的——这意味着需要另寻他处，找到编码指南。建议你熟悉下列资源，并在阅读本书和继续前端开发的过程中养成经常查阅文档的习惯。

Mozilla开发者网络（MDN）是HTML、CSS和JavaScript最好的参考文档，可以通过devdocs.io访问。这是一个优秀的文档界面（如图1-12所示），它 从MDN上拉取前端核心技术文档，而且还能在离线状态下查阅。

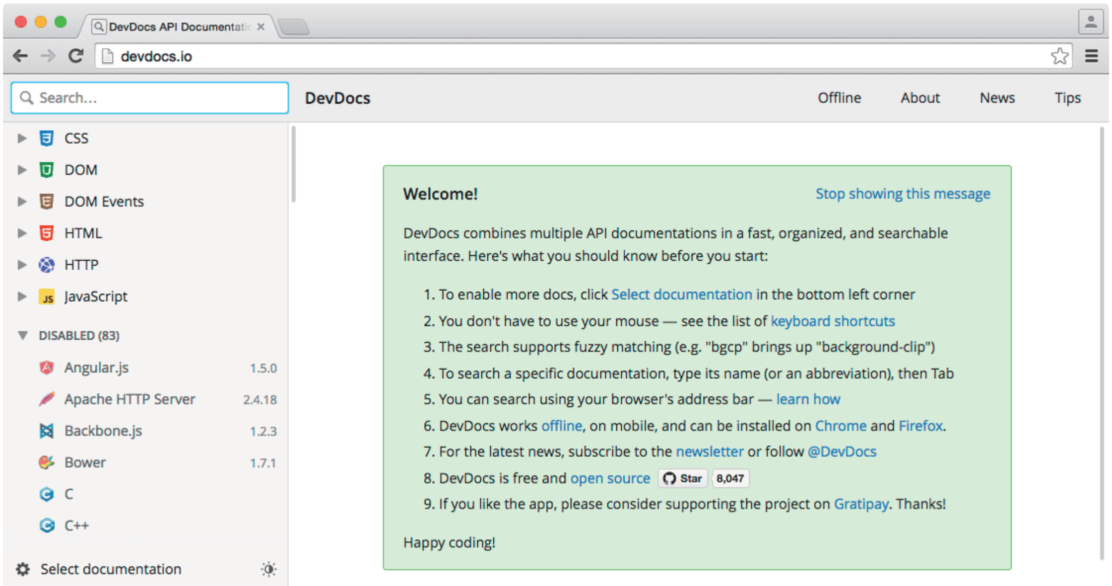


图1-12　通过devdocs.io访问文档

注意，Safari目前尚不支持devdocs.io所用到的离线缓存机制；如需访问，可使用Chrome这样的浏览器。

还可以访问MDN的官方网站developer.mozilla.org/en-US（如图1-13所示），或在搜索引擎中搜索“MDN”以获取所需信息。



图1-13　MDN官网

另一个需要知道的网站是stackoverflow.com（如图1-14所示）。严格说来，它并非文档源，而是一个开发者讨论社区。问题答案的质量视情况而定，但通常非常全面，很有帮助。这是很有用的资源，但需要记住，因其众包特性，答案并不具有权威性。

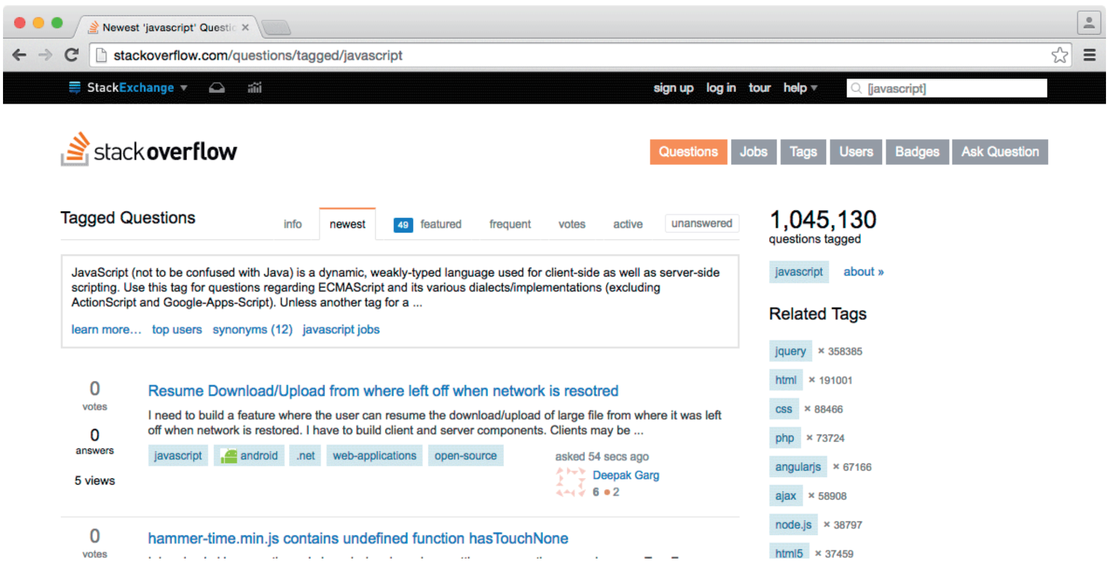


图1-14　Stack Overflow网站

Web技术在不断演进。随着时间推移，不同浏览器对特性和API的支持会有所不同。有两个

网站可以帮你判断某个特性被哪些浏览器（以及该浏览器的哪些版本）所支持，它们是html5please.com和caniuse.com。需要查询特性支持程度时，建议使用html5please.com查询该特性是否被推荐使用；需要了解浏览器的哪些版本支持某一特性时，则可以访问caniuse.com。

1.4　命令行速成

命令行（或称为**终端**）的使用贯穿全书，其中用到的许多工具都只会在命令行中运行。

要在Mac上访问命令行，需要打开Finder，依次进入Applications、Utilities文件夹，找到并打开名为Terminal的程序（如图1-15所示）。

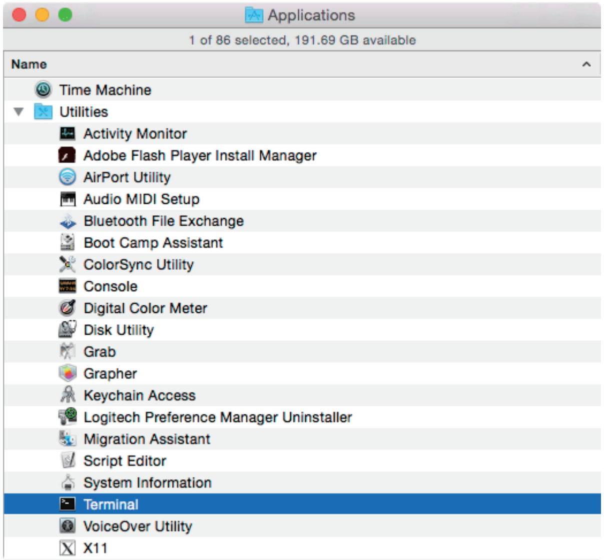


图1-15　在Mac上找到Terminal程序

然后会看到如图1-16所示的窗口。

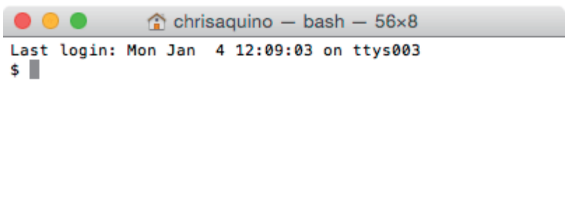


图1-16　Mac命令行

要在Windows上访问命令行，请打开“开始”菜单，搜索“cmd”，找到并打开名为Command Prompt的程序（如图1-17所示）。

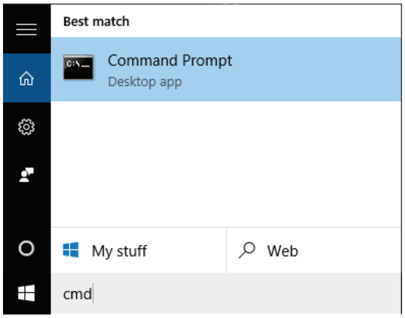


图1-17　在Windows上找到Command Prompt程序

点击即可运行Windows下的标准命令行界面，如图1-18所示。

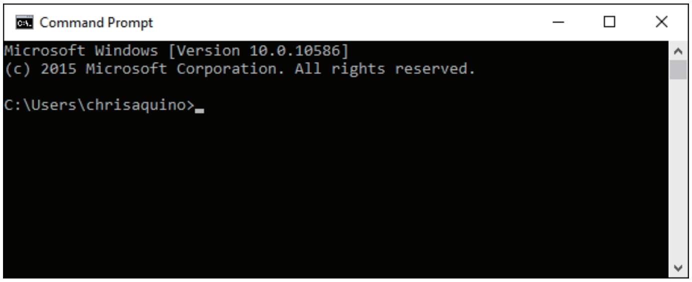


图1-18　Windows命令行

从现在起，将用“终端”或“命令行”统一指代Mac的Terminal和Windows的Command Prompt。若不熟悉命令行的使用，下面会介绍一些常见任务的命令。在窗口中输入命令后，按下回车键执行命令——所有命令都是如此。

1.4.1　查看当前工作目录

命令行是以当前位置为基础的，这意味着在任意给定时间点上，它总会“位于”文件结构的特定目录中，所有输入的命令都在该目录中执行。命令行窗口显示了当前目录的缩写。要在Mac上查看完整路径，请输入`pwd`（即print working directory，打印工作目录）命令，如图1-19所示。

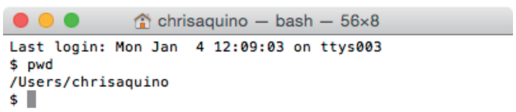


图1-19　在Mac上使用pwd展示当前路径

在Windows上使用`echo %cd%`命令查看当前路径，如图1-20所示。

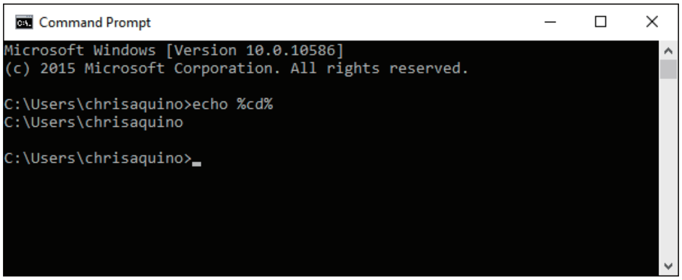


图1-20　在Windows上使用`echo %cd%`展示当前路径

1.4.2　新建目录

前端项目的目录结构非常重要。项目的增长可能很快，所以最好从一开始就保持良好的结构。在开发过程中会经常创建新目录，通过`mkdir`（即make directory，创建目录）命令加上新目录名称即可新建目录。

下面就为本书将要创建的所有项目新建一个目录。输入以下命令：

```
mkdir front-end-dev-book
```

紧接着，为第一个项目Ottergram新建一个目录，该项目会在下一章介绍。要创建的目录是刚新建的front-end-dev-book的子目录。因此，在主目录中的新目录名前面加上父级目录名以及一个斜杠（在Mac上）：

```
mkdir front-end-dev-book/ottergram
```

在Windows上，请使用反斜杠：

```
mkdir front-end-dev-book\ottergram
```

1.4.3　切换目录

要在文件结构中切换目录，请使用`cd`（即change directory，切换目录）命令，其后加上要进入的目录路径。

使用`cd`命令时，不用每次都输入完整目录路径。例如，要进入当前目录的某个子目录时，只需该子目录名称即可。当处于front-end-dev-book目录时，ottergram目录的路径就是ottergram。

进入新项目目录：

```
cd front-end-dev-book
```

现在进入ottergram目录：

```
cd ottergram
```

要进入父级目录，使用命令`cd ..`（即`cd`后面加上一个空格和两个句点）。两个句点代表父级目录的路径。

```
cd ..
```

记住，可以通过pwd（在Windows上是echo %cd%）命令查看当前目录。图1-21展示了作者新建目录、切换目录和查看当前目录的操作。

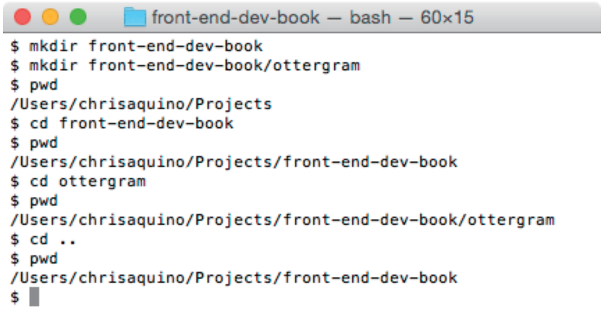


图1-21　切换、查看目录

每次切换目录不限于向下或向上一层。假设有一个结构更复杂的目录，如图1-22所示。

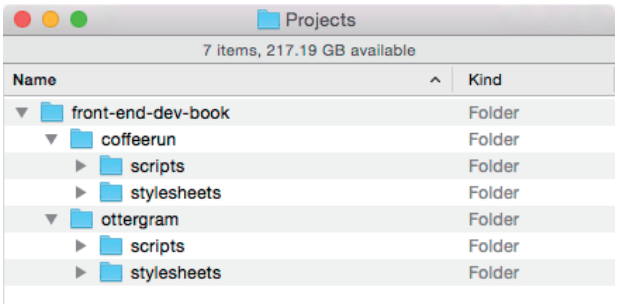


图1-22　文件结构示例

假设你现在在ottergram目录中，想要切换到coffeerun目录下的stylesheets。这可以通过在cd后面加上一个路径实现，该路径表示“当前所在目录的父级目录下的coffeerun目录中的stylesheets目录”：

```
cd ../coffeerun/stylesheets
```

在Windows上，命令是相同的，但要用反斜杠：

```
cd ..\coffeerun\stylesheets
```

1.4.4　列出目录中的文件

有时可能需要查看当前目录下有哪些文件，这在Mac上可以使用ls命令实现（如图1-23所示）。若想查看另一个目录下的文件列表，可以在命令后面加上相应路径：

```
ls
ls ottergram
```

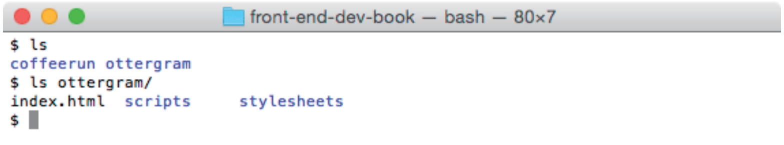


图1-23　使用ls列出目录中的文件

若目录为空，`ls`默认不会打印任何内容。

在Windows上，对应的命令是`dir`（如图1-24所示），也可以加上可选路径：

```
dir
dir ottergram
```

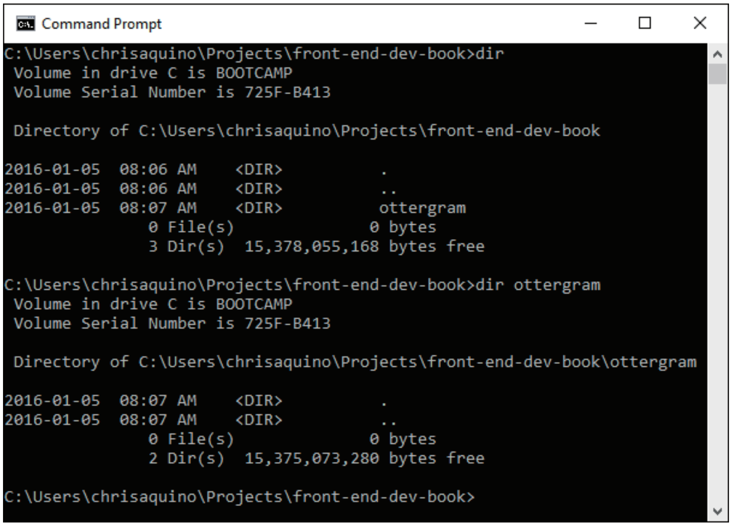


图1-24　使用dir列出目录中的文件

`dir`命令会默认打印出日期、时间、文件大小等信息。

1.4.5　获取管理员权限

在某些版本的OS X和Windows上，可能需要**超级用户**或管理员权限才能执行一些命令（如安装软件、修改受保护文件等）。

在Mac上，可以在命令前加上`sudo`获取权限。首次使用`sudo`时，会有如图1-25所示的警告出现。

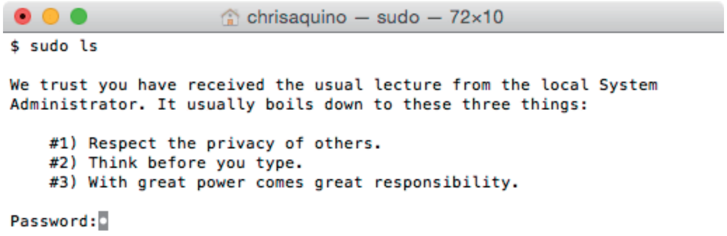


图1-25　sudo警告

sudo以超级用户身份执行命令之前会向你询问密码。输入密码时不会显示输入的内容，因此要小心，别输错。

在Windows上获取权限是在打开命令行界面的过程中完成的。在Windows的开始菜单中找到命令行选项，鼠标右键点击，选择Run as Administrator（如图1-26所示）。该命令行窗口中的所有命令都是以超级用户的身份运行的，因此同样要当心。

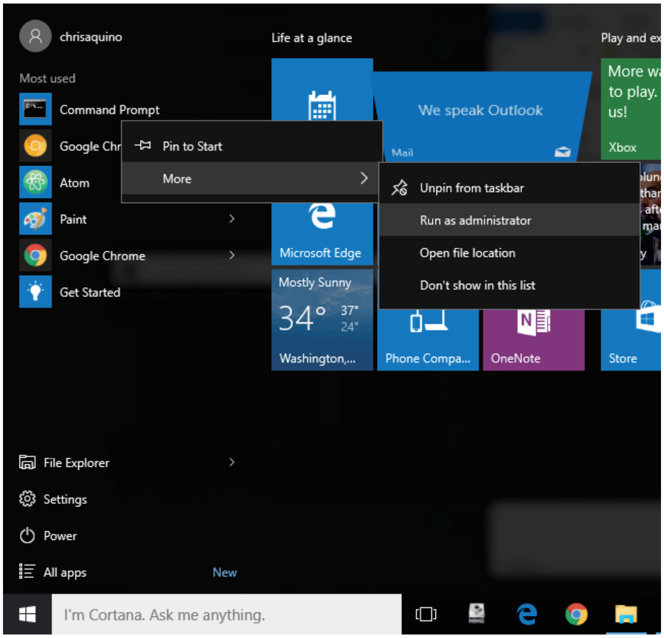


图1-26　以管理员身份打开命令行窗口

1.4.6　退出程序

在学习本书的过程中，你会通过命令行运行很多应用。有些应用在完成任务后会自动退出，但也有些应用在手动终止之前会保持运行。要退出命令行程序，使用Control + C快捷键。

1.5　安装 Node.js 和 browser-sync

这是开始第一个项目之前的最后一个安装步骤了。

Node.js（简称Node）允许通过命令行运行用JavaScript编写的程序。大多数前端开发工具都要使用Node.js。第15章会介绍更多关于Node.js的内容，但眼下要用到一个依赖Node.js的工具——browser-sync。

若要安装Node，需要从nodejs.org（如图1-27所示）下载安装包。本书使用的是Node.js 5.11.1版本，你看到的版本可能有所不同。



图1-27　下载Node.js

双击安装包，根据提示操作。

Node自带两个命令行程序：node和npm。node的工作是运行JavaScript程序，在第15章之前还用不到它；而另一个程序npm则会在在线安装开源开发工具时用到。

browser-sync对本书的价值不可估量。有了它，示例代码在浏览器中的运行将更加方便；修改并保存代码时，浏览器也会自动重新加载。

在命令行中通过下面的命令安装browser-sync：

```
npm install -g browser-sync
```

（命令行中的`-g`表示global，即全局。以全局方式安装此工具，意味着可以在任意文件夹中执行`browser-sync`命令。）

运行此命令时所处的目录不影响结果，但可能需要超级用户权限。若如此，在Mac的命令前加上`sudo`：

```
sudo npm install -g browser-sync
```

若在Windows上，则先以管理员身份打开命令行窗口（前文已提过）。

从下一章启动browser-sync后它便会一直运行，使用Control + C才能退出。完成一个项目之后，最好退出browser-sync。在本书的前两个项目（Ottergram和CoffeeRun）中，开始工作之前需要启动browser-sync。

通过前面的介绍和操作，Ottergram项目所需的工具已经准备好啦！

1.6　延展阅读：Atom 的替代工具

可供选择的文本编辑器太多太多。如果对Atom不那么感兴趣，在跟着本书完成所有项目之后，可以试试下面两个文本编辑器。它们在Mac和Windows平台上都可以免费获取，而且都拥有大量插件，可以个性化开发环境。此外，它们和Atom一样，都是由HTML、CSS和JavaScript构建，但以桌面应用的形式运行。

Visual Studio Code是微软为Web应用开发量身定做的开源文本编辑器，可通过code.visualstudio.com下载（如图1-28所示）。

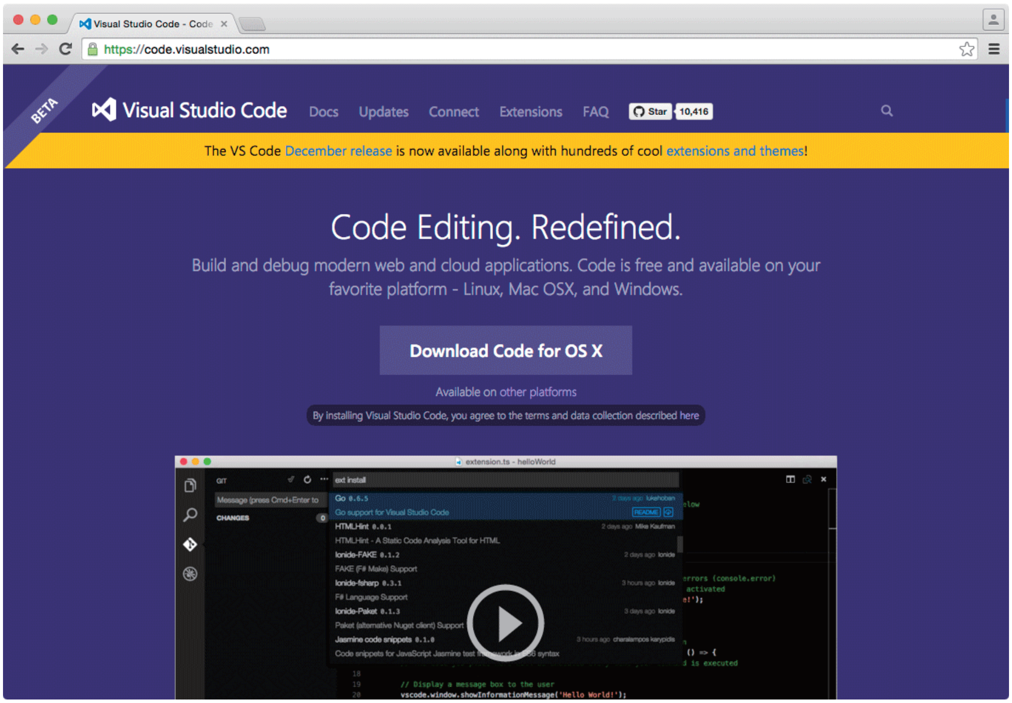


图1-28　Visual Studio Code官网

Adobe的Brackets文本编辑器专长于用HTML、CSS搭建UI。它还提供了一个扩展程序，帮助用户使用Adobe PSD文件。可以通过brackets.io下载Brackets（如图1-29所示）。

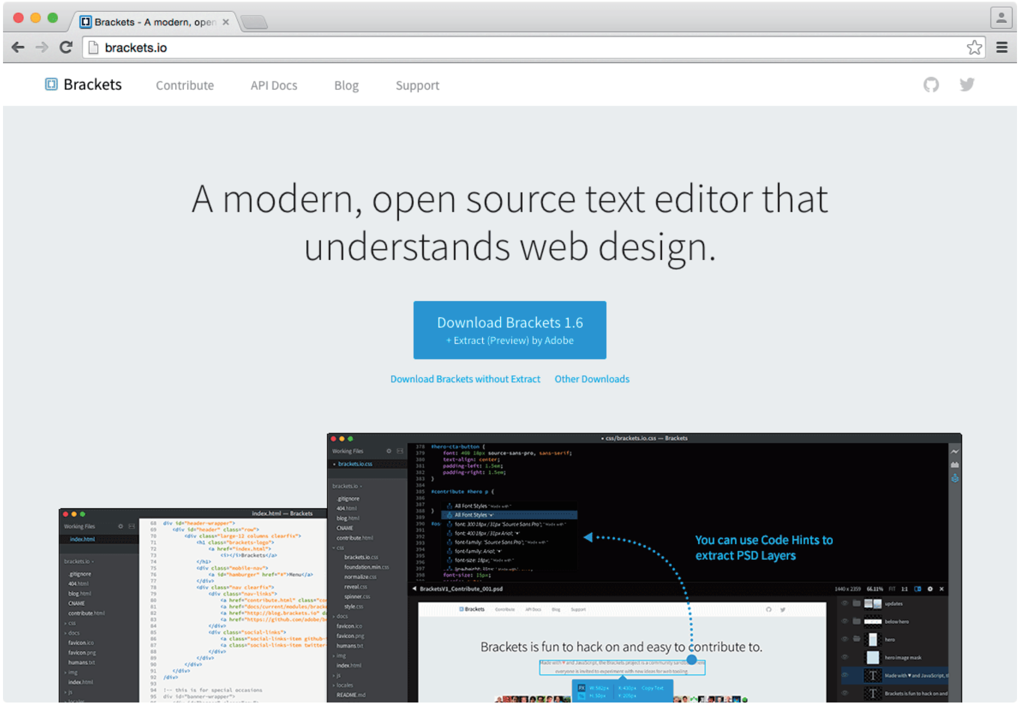


图1-29　Adobe Brackets官网

第 2 章 开始第一个项目

访问网站时，浏览器与服务器之间会进行对话——所谓的服务器其实就是互联网上的另一台电脑。

浏览器："你好！能给我cat-videos.html文件的内容吗？"

服务器："当然，让我找找看……找到啦！"

浏览器："它告诉我还需要另外一个叫styles.css的文件。"

服务器："好，我再找找看……找到啦！"

浏览器："好吧，这个文件又告诉我还需要一个文件， 它叫animated-background.gif。"

服务器："没问题，我再找找啊……找到啦！"

对话在一段时间内持续进行，有时会持续数千毫秒（如图2-1所示）。

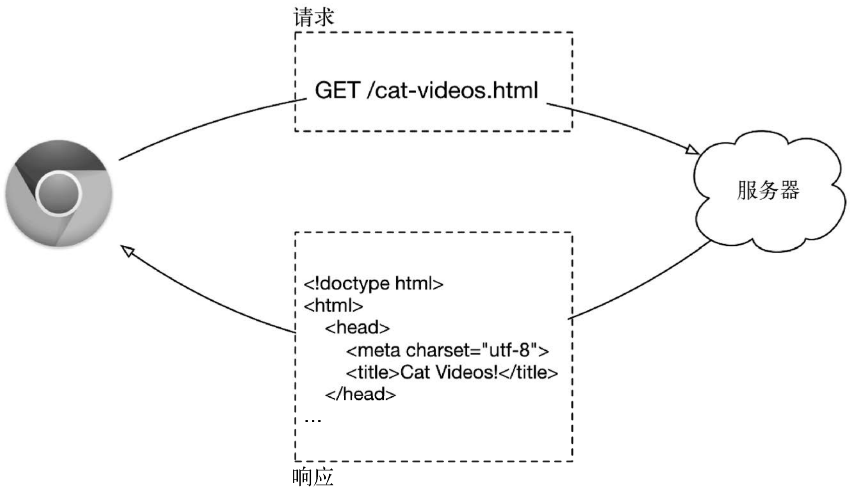


图2-1　浏览器发出请求，服务器响应

浏览器的工作是向服务器发送请求，解释从服务器收到的HTML、CSS和JavaScript，再将结果呈现给用户。一个网站的用户体验与HTML、CSS和JavaScript都有莫大的关系。假若将网站比作生物，HTML就是骨骼与器官（结构），CSS是皮肤（可视层），而JavaScript则是其个性（行为举止）。

本章将使用基本的HTML搭建第一个项目Ottergram。下一章开始使用CSS，第4章则用其进一步提升体验。到第6章，再加入JavaScript。

2.1　搭建 Ottergram

上一章为本书所有项目新建了目录，同时还给Ottergram项目新建了一个目录。启动Atom编辑器，点击File → Open（在Windows 上是File → Open Folder）打开ottergram文件夹。在对话框中，找到front-end-dev-book文件夹并选择ottergram。点击Open，告诉Atom使用该文件夹（如图2-2所示）。

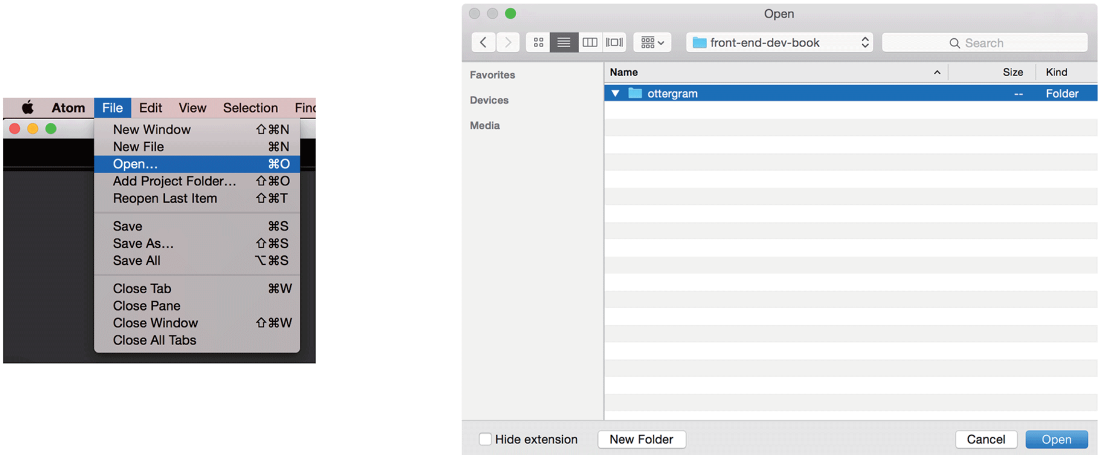


图2-2　在Atom中打开项目文件夹

在Atom左侧面板中可以看到ottergram文件夹，这块面板用于在项目文件和文件夹之间切换。

接着，通过Atom在ottergram文件夹下新建一些文件和文件夹。按住Control键，右击左侧面板的ottergram，在弹出菜单中点击New File选项。然后会弹出输入新文件名的文本框，输入index.html并敲下回车键（如图2-3所示）。

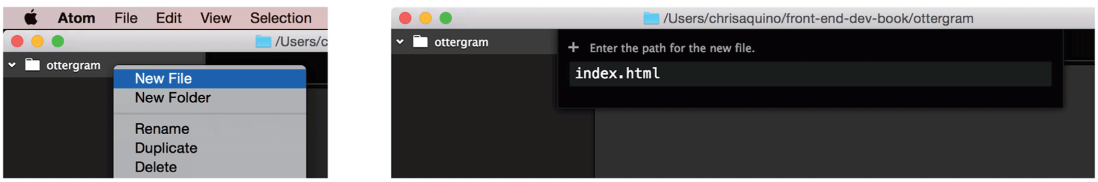


图2-3　在Atom中新建文件

用Atom新建文件夹的过程也是一样。再次按住Control键，右击左侧面板的ottergram，这次在弹出菜单中点击New Folder选项，然后在弹框中输入stylesheets（如图2-4所示）。

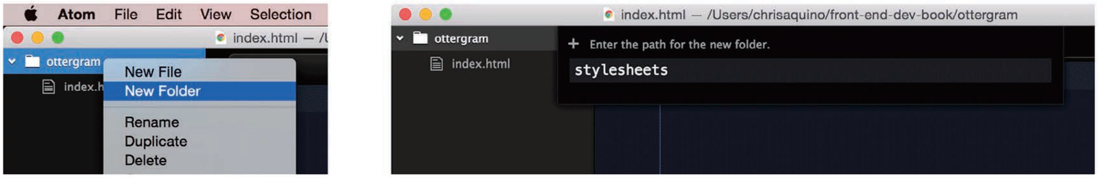


图2-4　在Atom中新建文件夹

最后，在stylesheets文件夹中新建名为styles.css的文件：按住Control键，右击左侧面板中的stylesheets并选择New File，弹框中会自动填充stylesheets/，然后输入styles.css并敲下回车键（如图2-5所示）。

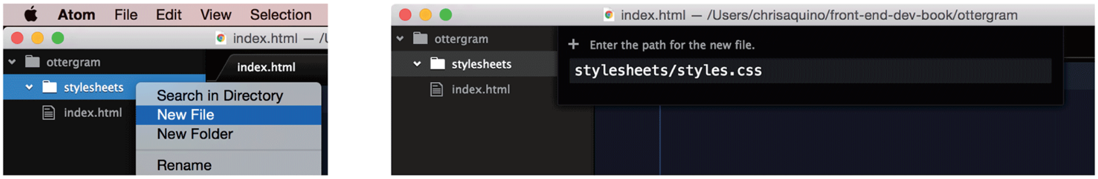


图2-5　在Atom中新建CSS文件

完成上述步骤之后，项目文件夹应该如图2-6所示。

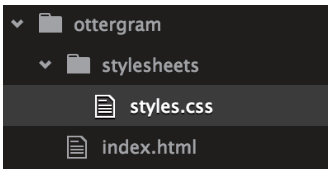


图2-6　Ottergram项目初始结构

如何组织、命名文件和文件夹并无定则，不过Ottergram（和本书中其他项目一样）遵循众多前端开发者约定俗成的，在index.html中存放HTML代码。将HTML主文件命名为index.html的习惯可以追溯到Web发展早期，这项约定延续至今。

stylesheets文件夹正如其名，可以存放一个或多个Ottergram项目的样式信息（即Cascading Style Sheet，CSS，层叠样式表）文件。有时，开发者会根据文件在页面/站点中适用的部分来为其命名，如header.css、blog.css等。Ottergram是一个简单项目，只需要一个CSS文件，将其命名为styles.css以表明其全局的角色。

2.1.1　开始写HTML

是时候动笔写代码了。在Atom中打开index.html，添加一些基本的HTML。

首先输入html，Atom会给出自动补全提示，如图2-7所示。（若没有提示，请检查是否已按照

第1章的指导安装了emmet插件。）

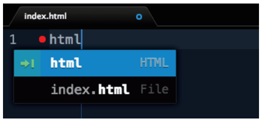


图2-7　Atom的自动补全选项

按下回车键，Atom会给出HTML元素基本框架（如图2-8所示）。



图2-8　自动补全功能创建的HTML

现在，光标在文档标题的起始标签`<title>`和闭合标签`</title>`之间。输入ottergram，给项目取一个名字；然后将光标移到`<body>`标签和`</body>`标签中间的空行，输入header并按下回车键，Atom会将header转换为`<header>`标签和`</header>`标签，中间有一行空行（如图2-9所示）。



图2-9　自动补全功能创建的header标签

接下来，输入h1并按下回车键——它同样会被转换成标签，但这次没有空行。再次输入ottergram，这是将显示在网页中的标题。

这样一来，文件应如下所示：

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
```

```
    <title>ottergram</title>
  </head>
  <body>
    <header>
      <h1>ottergram</h1>
    </header>
  </body>
</html>
```

Atom和emmet帮我们节省了不少打字的时间，生成了结构良好的HTML代码。

来看看代码。第一行的<!doctype html>定义了doctype（**文档类型**），告诉浏览器当前文档是用哪个版本的HTML编写的。doctype不同，浏览器渲染/绘制的页面可能也略有不同。在这里，doctype的意思是HTML5。

更早的HTML版本的doctype通常又长又复杂，不便于记忆，比如下面这个：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

这样一来，每次新建文档都得查一下doctype。

HTML5中的doctype变短了，看起来也更舒服。本书中的项目将全部使用这种doctype，你的项目也应该使用它。

doctype之后是包含head和body的一些基本HTML标记。

head中包含文档信息，以及浏览器如何处理文档的相关信息，比如说文档标题、页面使用的CSS/JavaScript文件、文档上次修改的时间等都包括在head中。

head中的<meta>标签为浏览器提供文档自身信息，如文档作者姓名、搜索引擎关键词等。Ottergram项目中的<meta>标签<meta charset="utf-8">指定文档由包含所有Unicode字符的UTF-8字符集进行编码。请在文档中使用该标签，以便多数浏览器能正确地解释代码，特别是在想获得更多国际流量的情况下。

body中包含所有代表页面内容的HTML代码：页面中出现的所有图片、链接、文本、按钮、视频等。

多数标签中还包含其他内容。看看刚加入的h1标题，其结构如图2-10 所示。

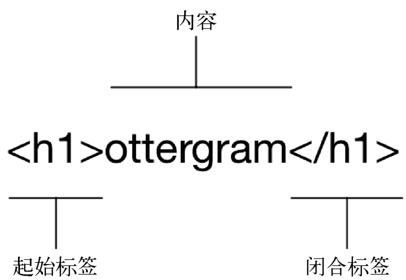


图2-10 简单HTML标签结构示意

HTML指的是“超文本标记语言”（Hypertext Markup Language）。标签就是用来“标记”内容的，并且还标明其用途（如标题、列表项、链接等）。

一组标签中的内容同样可以包含其他HTML。注意看，在上面的代码中，`<header>`包裹着`<h1>`标签。（还有`<body>`中包裹着`<header>`！）

可供选择的标签有很多，超过140个。可以通过MDN的HTML element参考文档查看，地址是developer.mozilla.org/en-US/docs/Web/HTML/Element。参考文档按照用途（如文本内容、内容分节、多媒体等）对元素进行分组，每一种元素都附有简介。

2.1.2　链接到样式表

第3章将在样式表文件styles.css中编写样式规则。不过，还记得本章开头浏览器与服务器之间的对话吗？只有当浏览器被告知文件存在时，它才会向服务器请求该文件。所以你需要**链接**样式表，告诉浏览器去请求它。更新index.html中的`head`部分，加入styles.css文件的链接。

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>ottergram</title>
    <link rel="stylesheet" href="stylesheets/styles.css">
  </head>
  <body>
...
```

通过`<link>`标签，可以给HTML文档附加外部样式表。它有两个**属性**，向浏览器提供更多关于该标签用途的信息（如图2-11所示）。（HTML属性的顺序不重要。）

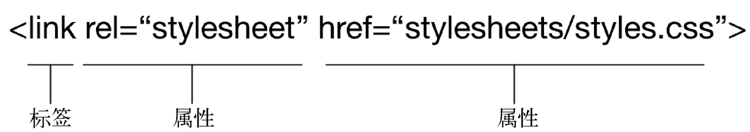


图2-11　带属性的标签结构

将`rel`（即relationship）属性设置为`"stylesheet"`，让浏览器知道链接的文档提供的是样式信息。`href`属性告诉浏览器向服务器请求stylesheets文件夹下的styles.css文件。注意这里的文件路径是**相对**于当前文档的。

往后看之前，记得先保存index.html。

2.1.3　添加内容

没有内容的网页，就像没有咖啡喝的日子。在`header`头部之后添加一个列表吧，列出该项目存在的理由。

接下来要添加一个**无序列表**，标签是`<ul>`。列表中包含5个`<li>`列表项，每一项中是`<span>`标签包裹的文本。

更新过的index.html如下所示。注意，本书中新加入的代码都会用粗体标出，需要删除的代码会用线条划去，原有代码以普通样式显示，以便读者更容易看到有变动的地方。

强烈建议充分使用Atom的自动补全和自动格式化功能。移动光标，输入ul，并按下回车键；然后输入li，并回车两次；接着输入span，并回车；输入一只水獭的名字，并以同样方式重复4次以创建另外4项。

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>ottergram</title>
    <link rel="stylesheet" href="stylesheets/styles.css">
  </head>
  <body>
    <header>
      <h1>ottergram</h1>
    </header>
    <ul>
      <li>
        <span>Barry</span>
      </li>
      <li>
        <span>Robin</span>
      </li>
      <li>
        <span>Maurice</span>
      </li>
      <li>
        <span>Lesley</span>
      </li>
      <li>
        <span>Barbara</span>
      </li>
    </ul>
  </body>
</html>
```

`<li>`标签中嵌套的`<span>`标签并无任何特殊意义，只是其他内容的普通容器，在Ottergram项目中使用它们是出于样式目的。随着本书的展开，你将会看到其他的容器元素。

再往下，根据水獭的名字添加图片。

2.1.4 添加图片

本书所有项目用到的资源都可以通过地址www.bignerdranch.com/downloads/front-end-dev-resources.zip下载，其中包括从commons.wikimedia.org获取的5张水獭照片，由Michael L. Baied、Joe Robertson和Agunther等人拍摄，采用创作共用授权。

下载、解压资源，在ottergram-resources文件夹中找到img文件夹，将它复制到ottergram项目目录中。（这个zip文件中还包括其他资源，但目前只需要img文件夹。）

除标题之外，你还希望列表中包含可点击的缩略图。可以给ul下的每一项加上锚链接和图片标签，到时候我们会详细说明。（如果使用了自动补全功能，注意调整标签的位置，使其紧跟在之后。）

```
...
    <ul>
      <li>
        <a href="#">
          <img src="img/otter1.jpg" alt="Barry the Otter">
          <span>Barry</span>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img/otter2.jpg" alt="Robin the Otter">
          <span>Robin</span>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img/otter3.jpg" alt="Maurice the Otter">
          <span>Maurice</span>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img/otter4.jpg" alt="Lesley the Otter">
          <span>Lesley</span>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img/otter5.jpg" alt="Barbara the Otter">
          <span>Barbara</span>
        </a>
      </li>
    </ul>
...
```

如果代码缩进不是很整齐，可以借助此前安装的atom-beautify插件。点击Packages → Atom Beautify → Beautify，然后代码就会缩进并对齐。

来看看又新增了哪些东西。

<a>是**锚**标签。锚标签使页面元素可点击，从而将用户带到另一个页面。它们通常被称为“链接”，但请注意，这和之前用到的<link>标签完全是两回事。

锚标签有一个名为href的属性，用于表明锚所指向的资源，其值通常是一个Web地址。但有时候你并不想去别的地方（这里就是如此），所以将“空”值#赋给href属性。这样一来，点击图片时页面就会滚动到顶部。稍后会实现点击缩略图后打开大图的效果。

我们在锚标签内部添加了`<img>`，即**图片**标签。它拥有`src`属性，值为之前添加的img目录中的文件名；还添加了描述性的`alt`属性，当无法加载图片时，`alt`属性会显示其所含文本。此外，读屏软件可以将`alt`文本作为图片描述告知视障用户。

与多数标签不同，`<img>`标签不会包裹其他元素，而是指向某个资源。当浏览器遇到`<img>`标签时，会在页面上绘制图片，这就是所谓的**可替换元素**；嵌入的文档和Java小程序也是可替换元素。

因为不包裹任何内容或元素，`<img>`标签不存在对应的闭合标签，故被称为**自闭合**标签（或**空**标签）。有时你会发现`<img src="otter.jpg" />`这样的写法，即右尖括号之前多了一个斜杠。其实加不加斜杠只是个人偏好问题，对浏览器来说没有区别。本书使用不带斜杠的自闭合标签。

保存index.html，很快就能看到代码的结果啦！

2.2　浏览网页

要查看网页，就需要运行第1章安装的browser-sync工具。

打开命令行，切换工作目录至ottergram文件夹——想想在第1章中是如何使用`cd`命令加目标文件夹路径切换目录的。获取ottergram路径的一个便捷方法是按住Control键，右击Atom左侧面板的ottergram文件夹，选择Copy Full Path（如图2-12所示）。然后在命令行中输入`cd`，粘贴路径，按下回车。

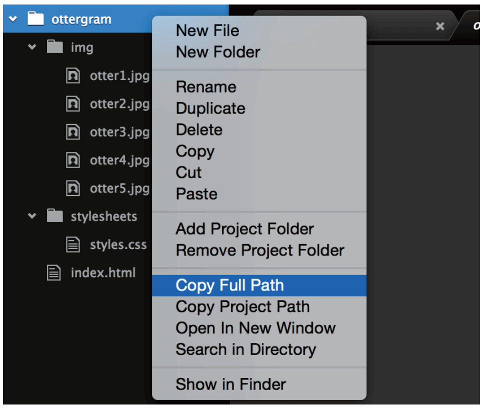


图2-12　在Atom中复制ottergram文件夹的路径

你所输入的路径应该如下：

```
cd /Users/chrisaquino/Projects/front-end-dev-book/ottergram
```

切换完成之后，运行下面的命令，以便在Chrome上打开Ottergram。（为适应页面将命令分为两行，实际应当在一行中输入所有内容。）

```
browser-sync start --server --browser "Google Chrome"
                   --files "stylesheets/*.css, *.html"
```

若Chrome已经是默认浏览器，则可以去掉`--browser "Google Chrome"`这部分：

```
browser-sync start --server --files "stylesheets/*.css, *.html"
```

该命令以服务器模式启动browser-sync，它会在浏览器请求文件时发送响应（比如当创建index.html文件时）。

以上命令同时还告诉browser-sync，当HTML或CSS文件发生改动时，自动重新加载页面——这大大提高了开发效率。在browser-sync这一类工具出现之前，每次变动都需要手动刷新页面。

图2-13展示了在Mac上输入命令的结果。

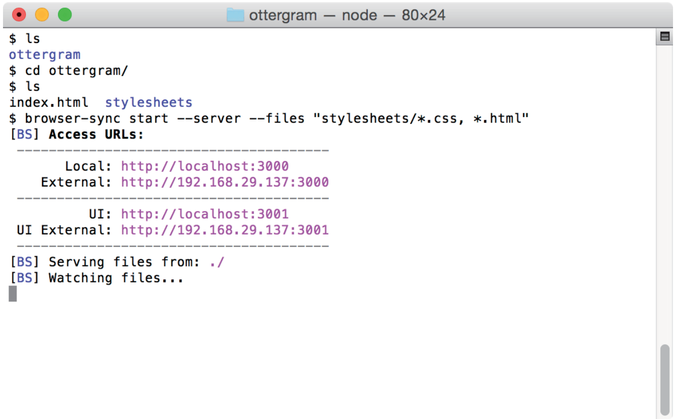


图2-13　在Mac命令行终端中启动browser-sync

在Windows上也能看到相同输出（如图2-14所示）。

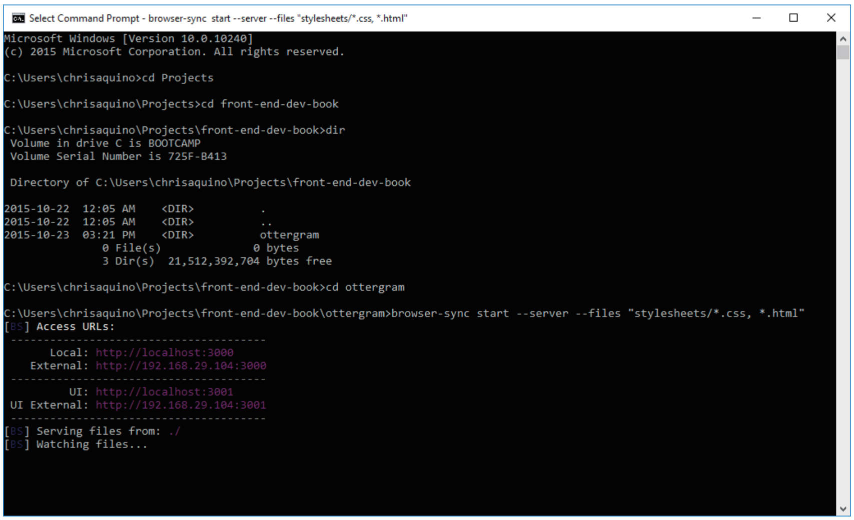


图2-14　在Windows命令行终端中启动browser-sync

在Chrome中成功加载Ottergram页面以后便可以看到页面中的ottergram标题、标签页中的ottergram以及一系列水獭图片和名字（如图2-15所示）。

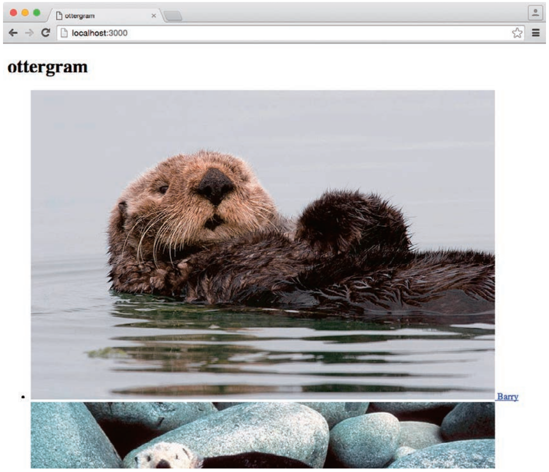


图2-15 在浏览器中查看Ottergram

2.3 Chrome 开发者工具

用Chrome内置的开发者工具（通常被称为DevTools）来调试样式和布局等再好不过了，通过开发者工具调试比在代码中试验高效很多。DevTools非常强大，是你前端开发之路上的忠实伙伴。

下一章开始才会用到开发者工具。现在先打开窗口，熟悉主要区域。

点击地址栏右侧的≡图标，选择More Tools → Developer Tools（如图2-16所示）。

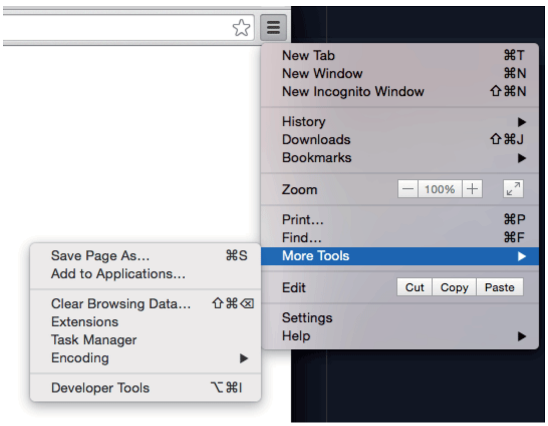


图2-16 打开开发者工具

开发者工具默认在右侧显示。这时屏幕看起来应该如图2-17 所示。

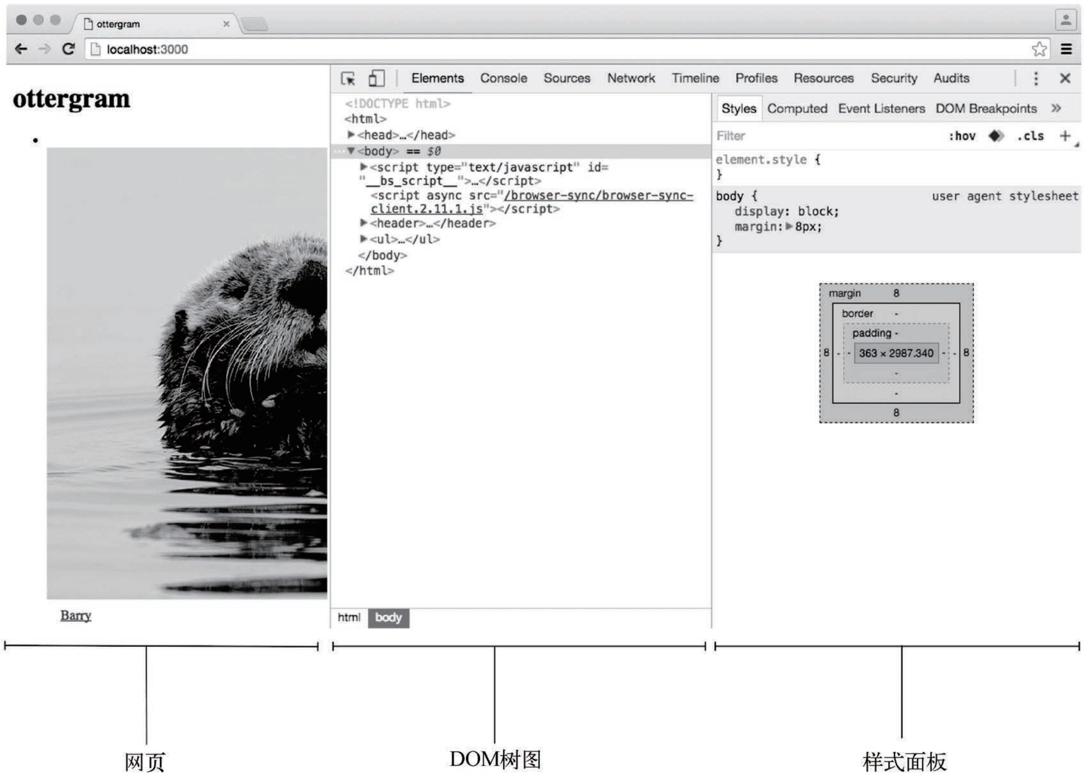


图2-17　开发者工具的Elements面板

开发者工具展现了代码与其生成的页面元素之间的关系，通过它能检查单个元素的属性、样式，也能即时看到浏览器是如何解释代码的。这些关系对开发和调试都至关重要。

由图2-17可知，网页右侧的开发者工具显示的是Elements面板，该面板又可以分为两部分：左侧是DOM树图，代表着解释为DOM元素（Document Object Model，即文档对象模型，本书后面会介绍更多相关内容）的HTML；右侧是样式面板，展示应用于元素的视觉样式。

在工作时，将开发者工具放在屏幕右侧通常比较方便。若需要改变其位置，请点击右上角的 ⋮ 按钮，在弹出菜单中有用于切换开发者工具位置的按钮（如图2-18所示）。

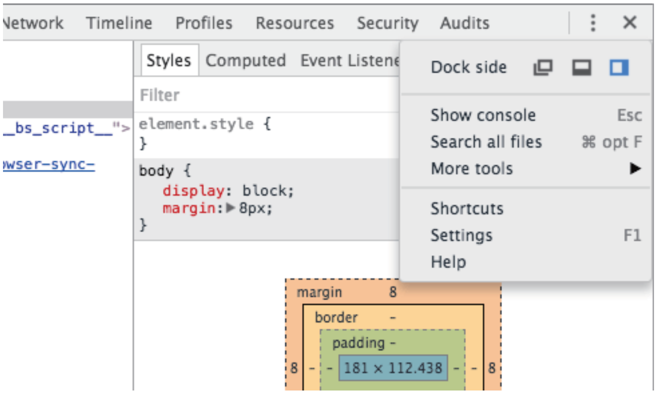


图2-18　切换开发者工具的位置

一切准备就绪，下一章开始添加样式。

2.4　延展阅读：CSS 版本

CSS历史版本包括标准版本1、2和2.1。CSS2.1之后，因其规模不断增长，标准被分成多个部分。

CSS版本3并不存在。所谓的CSS3只是对一系列模块的概括性称呼，每个模块都有自己的版本号。

表2-1　CSS版本：真实与想象

版本号	发布年份	显著特性
1	1996	基本的字体属性（`font-family`，`font-style`）、前景色与背景色、文本对齐、外边距、边框、内边距
2	1998	绝对定位、相对定位、固定定位；新增字体属性
2.1	2011	删除了一些很少有浏览器实现的特性
“3”	不定	一系列不同规范的集合，如媒体查询、新增选择器、半透明颜色、`@font-face`等

2.5　延展阅读：favicon.ico

你注意过在经常访问的网页的地址栏左侧有一个小图标吗？有时候这些图标也出现在浏览器标签页中，如图2-19所示。

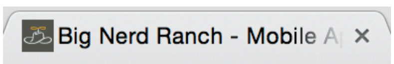


图2-19　bignerdranch.com的favicon.ico

这就是许多网站都有的favicon.ico图片文件，浏览器也会默认请求这种文件。因为Ottergram

项目还没有添加它，所以你可能在开发者工具中会注意到如图2-20所示的错误信息。

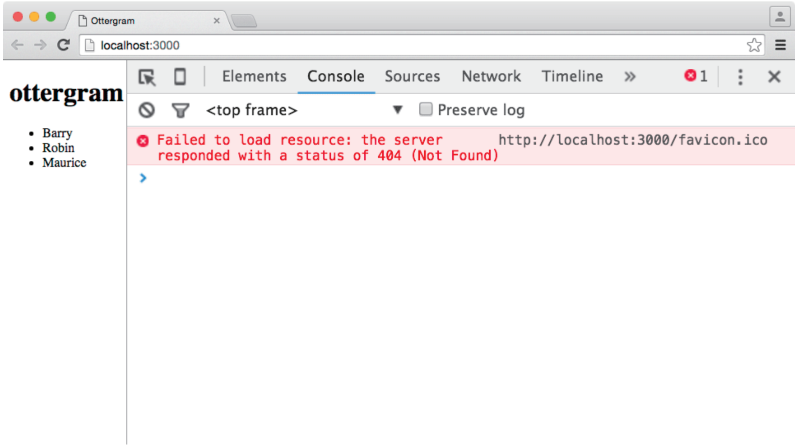


图2-20　因缺少favicon.ico而报错

如果出现错误，请不用担心，这不会对项目产生影响。favicon.ico添加起来很容易，这是我们遇到的第一个挑战。

2.6　中级挑战：添加 favicon.ico

相比于错误信息，你肯定更乐于看到水獭。试着用一张水獭照片制作自己的favicon.ico文件吧。

搜索“favicon generator”可以找到一堆能帮你转换文件的网站。它们多半需要你上传图片，然后就可为你提供一个自己的favicon.ico。

选择并上传一张水獭照片吧。

将favicon.ico文件保存到index.html所在的文件夹。最后，刷新浏览器，浏览器标签页现在应该如图2-21所示。



图2-21　为项目添加favicon.ico之后

第 3 章 样 式

这一章开始设计静态版的Ottergram项目，之后再添加交互。

本章完成后的网页效果如图3-1所示。



图3-1 加入样式后的Ottergram

本章将介绍一系列的概念与示例。若读完后感觉似乎并未掌握所有知识点，大可不必担心，因为在本书中你还会一次次地邂逅它们。本章的任务是为你理解后续内容打下坚实的基础。

当然，这里介绍的内容仅仅是CSS的皮毛，你可以查阅MDN了解所有属性。

前端开发者给网站添加样式时，常面临两种选择，是先整体后局部，还是先局部后整体。

从细节入手进而谋划大局能使代码更加整洁，复用性也更高。这种工作方式被称作**原子样式**

（atomic styling），本章使用的就是这样的方式——首先为缩略图添加样式，接着是列表布局，下一章则对网站整体进行布局。

3.1　创建基本样式

首先将normalize.css文件添加到项目中，normalize.css让CSS代码在不同浏览器上表现一致。每种浏览器都有一组默认样式，却不尽相同。要开发网站或应用的自定义样式，normalize.css是不错的起点。

可以在线获取免费的normalize.css。要将它添加到Ottergram也无须下载到本地，只需将链接添加到index.html即可。

为确保使用的normalize.css是最新版本，建议你从内容分享网站获取文件。访问cdnjs.com/libraries/normalize，找到以.min.css结尾的文件。（该文件删除了多余的空白字符，比其他版本要小一些。）点击Copy按钮复制地址（如图3-2所示）。

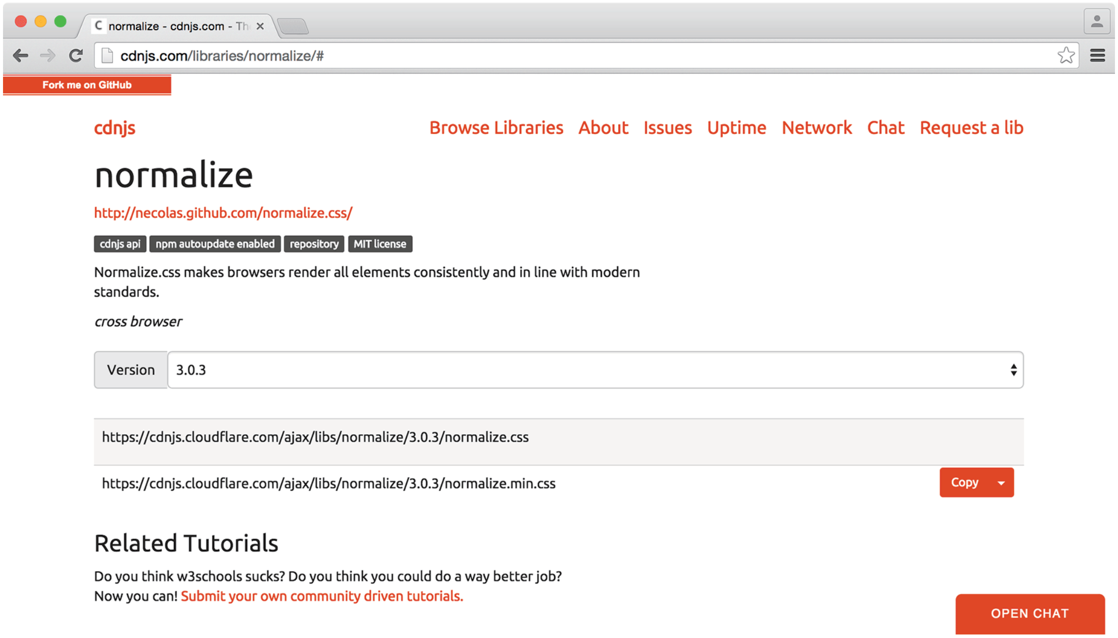


图3-2　从cdnjs.com获取normalize.css的链接

创作本书时的最新版本是3.0.3，目前的版本可能更高。

在Atom中打开ottergram文件夹，并打开index.html。添加<link>标签，粘贴刚刚复制的地址。（下面代码中的<link>标签被分为两行以适应排版，实际可以写在一行中。）

```
<!doctype html>
<html>
  <head>
```

```
    <meta charset="utf-8">
    <title>ottergram</title>
    <link rel="stylesheet"
      href="https://cdnjs.cloudflare.com/ajax/libs/normalize/3.0.3/normalize.min.css">
    <link rel="stylesheet" href="stylesheets/styles.css">
  </head>
...
```

请确保normalize.css的`<link>`标签在styles.css的`<link>`标签**之前**，浏览器需要先读取normalize.css中的样式，再读取自定义样式。

这样就完成了，不需要再配置其他东西了。

可能有人好奇，使用的地址怎么是其他服务器上的？实际上，对HTML文件来说，这再正常不过了（如图3-3所示）。

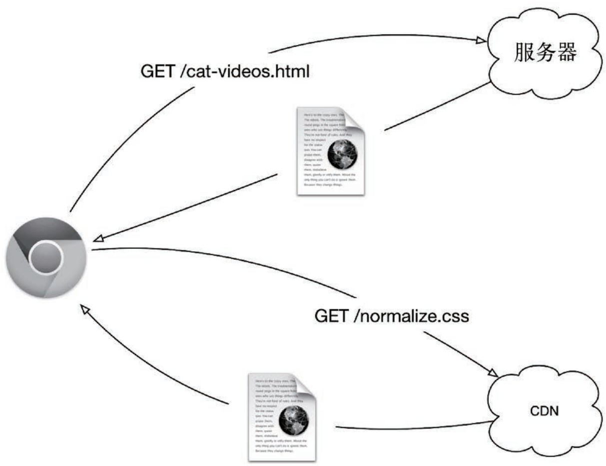


图3-3 从不同服务器请求资源

刚刚加入的normalize.css托管在cdnjs.com上，这是一个公共服务器，是**内容分发网络**（Content Delivery Network，CDN）的一部分。CDN在世界各地都有服务器，每台服务器上都有相同文件的副本。用户发出请求后，将通过最近的服务器返回文件，减少加载时间。cdnjs.com上托管了各种版本的前端流行库和框架。

3.2 为 HTML 文件添加样式

本章将在上一章创建的styles.css中加入一些CSS**样式规则**。添加样式之前，需要为HTML添加样式“钩子”。

首先，为显示水獭名字的`span`元素添加类名，将这些元素标记为“缩略图标题”。类名可用于标记一组HTML元素，通常用于添加样式。有了类名，为水獭名字添加样式就很方便了。

在index.html中为`li`元素中的`span`添加`thumbnail-title`类名属性，如下所示：

```
...
    <ul>
      <li>
        <a href="#">
          <img src="img/otter1.jpg" alt="Barry the Otter">
          <span>Barry</span>
          <span class="thumbnail-title">Barry</span>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img/otter2.jpg" alt="Robin the Otter">
          <span>Robin</span>
          <span class="thumbnail-title">Robin</span>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img/otter1.jpg" alt="Maurice the Otter">
          <span>Maurice</span>
          <span class="thumbnail-title">Maurice</span>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img/otter4.jpg" alt="Lesley the Otter">
          <span>Lesley</span>
          <span class="thumbnail-title">Lesley</span>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img/otter5.jpg" alt="Barbara the Otter">
          <span>Barbara</span>
          <span class="thumbnail-title">Barbara</span>
        </a>
    </ul>
...
```

很快就可以用刚添加的类名为所有图片标题添加样式了。

3.3　样式的构成

通过书写样式规则创建样式。样式规则包括两部分，**选择器**和**样式声明**，如图3-4 所示。

样式规则的第一部分是一个或多个选择器。选择器描述样式将应用于哪些元素，如h1、span或img等。不过可用作选择器的远不止标签名，你还可以书写一些能提高**优先级**的选择器，将样式应用于一组指向更明确的元素。

例如，可以使用属性作为选择器，如刚刚为<span>标签添加的thumbnail-title类名。类名选择器比标签选择器的优先级高。

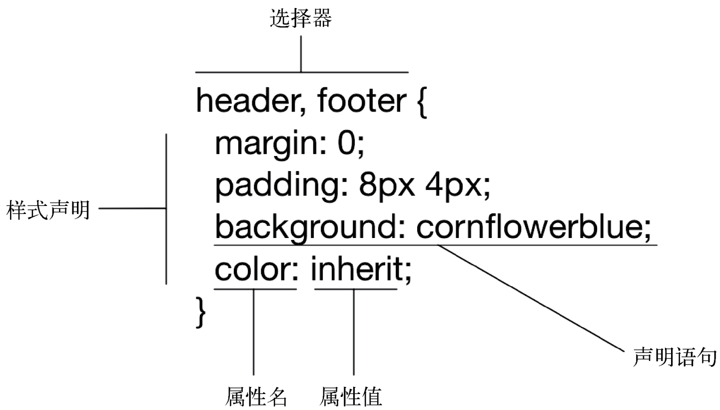


图3-4　样式规则的结构

优先级确保样式仅应用于特定的一组元素（试比较拥有`thumbnail-title`类名的元素与所有`<span>`元素），且决定了选择器的相对权重。若样式表包含应用于相同元素的多个样式，将会应用选择器优先级较高的样式。在本章末尾的延展阅读中，你可以了解到更多关于优先级的知识。

本章会介绍不同种类的选择器，它们在优先级方面有所不同。在样式中指向相同元素的方式虽然很多，但理解优先级才是使用最佳选择器、编写可维护样式的关键。

样式规则的第二部分是由大括号包裹着的样式声明，它定义要应用的样式。每一条声明由属性名和属性值构成。

编写第一条样式规则时，使用刚添加的类名作为选择器，为水獭们的名字添加样式。

3.4　第一条样式规则

要在样式规则中将类名作为选择器，只需在类名前加上句点即可，如`.thumbnail-title`。首先为`.thumbnail-title`类添加背景色和前景色。

打开styles.css，添加样式规则：

```
.thumbnail-title {
  background: rgb(96, 125, 139);
  color: rgb(202, 238, 255);
}
```

稍后会介绍更多关于颜色的知识，目前只需要看看发生的变化。保存styles.css，确保browser-sync已经运行。若需要重新启动browser-sync，请使用如下命令：

```
browser-sync start --server --browser "Google Chrome"
                   --files "stylesheets/*.css, *.html"
```

网页会在Chrome中打开，如图3-5所示。



图3-5　多了点色彩的Ottergram

可以看到，缩略图标题的背景颜色变成了深蓝色，而文字颜色变成了浅蓝色。赞！

继续为缩略图标题添加样式，如下：

```
.thumbnail-title {
  display: block;
  margin: 0;
  padding: 4px 10px;

  background: rgb(96, 125, 139);
  color: rgb(202, 238, 255);
}
```

新增的三条样式声明都会影响元素**盒子**。对于每个可见的HTML 元素，浏览器都会在页面中绘制一个矩形。浏览器使用一种名为**标准盒模型**（简称为“盒模型”）的方案决定这些矩形的大小。

盒模型

为理解盒模型，可以打开开发者工具查看一番。保存styles.css，切换到Chrome，并打开开发者工具（如图3-6所示）。

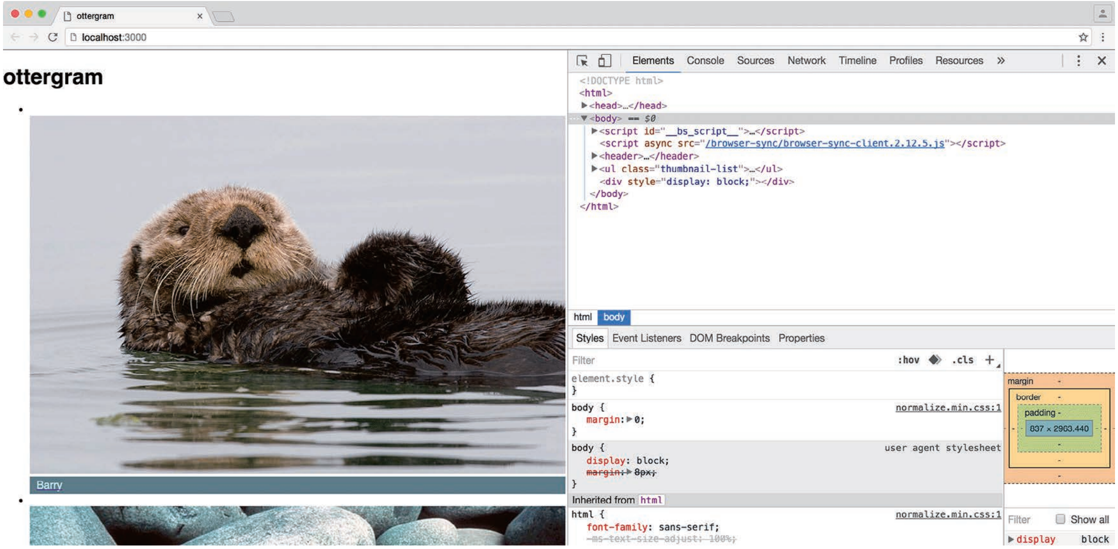


图3-6　探索盒模型

点击Elements面板左上角的按钮，即**检查元素**按钮，并将光标移到网页中的ottergram单词上。这时可以看到，标题周围出现蓝色和桃红色的矩形（如图3-7所示）。

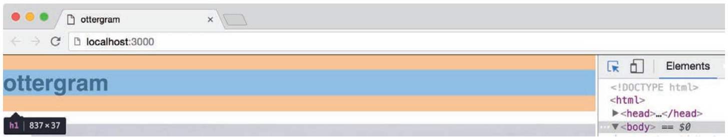


图3-7　光标悬停在标题上

点击单词ottergram后，几种颜色的矩形不见了，元素被选中，DOM树图展开，高亮显示相应的<h1>标签。

Elements面板右下方的矩形图展示了h1元素的盒模型。注意，图示某些部分的颜色与之前检查标题时看到的矩形颜色是相同的（如图3-8所示）。

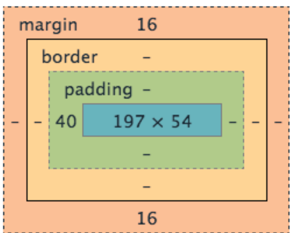


图3-8　查看元素的盒模型

一个元素的盒模型由四个部分组成（在矩形图示中，开发者工具以四种不同的颜色分别表示这四个部分）。

内容（蓝色）	可见内容，就是文本
内边距（绿色）	内容四周透明的部分
边框（黄色）	可将环绕内容和内边距的部分设置为可见
外边距（桃红色）	边框之外的透明部分

图3-8中的数值单位是**像素**，1像素对应屏幕中显示单一颜色的最小矩形区域。h1元素的内容区域为宽197像素、高54像素的区域（屏幕大小不同，看到的值也可能不同）。元素左侧有40像素的内边距，边框为0像素，上下外边距各为16像素。

这个外边距从何而来呢？每个浏览器都提供了**浏览器样式表**，在HTML文件没有指定的情况下，为HTML元素提供默认样式。因为我们尚未给h1元素盒子指定任何样式，所以使用默认样式。

这样一来，就可以理解前面添加的样式声明了：

```
.thumbnail-title {
  display: block;
  margin: 0;
  padding: 4px 10px;

  background: rgb(96, 125, 139);
  color: rgb(202, 238, 255);
}
```

display: block声明更改了所有类名为.thumbnail-title的元素的盒子，使这些元素可以占据包含它们的元素的完整宽度。（注意在图3-6中，缩略图标题的背景色覆盖了更宽的区域。）display属性的其他值会在之后讲到，如display: inline使元素宽度与内容相适应。

此外，我们还将缩略图标题的margin设为0，padding设为两组不同的值：4px和10px（px是像素的缩写）。指定具体的padding值，以覆盖浏览器默认值。

padding、margin和其他一些样式可以使用**简写属性**，即将一个值应用于多个属性。例如，当只给padding提供两个值时，第一个值作用于垂直方向（上下），第二个值作用于水平方向（左右）；还可以只提供一个值，作用于四个方向；也可以分别为四个方向指定值。

概括来说，新加入的样式声明意味着所有类名为.thumbnail-title的元素的盒子将会撑满容器宽度，没有外边距，上下padding为4px，左右padding为10px。

3.5　样式继承

下面来添加样式，修改文本大小和外观。

在CSS文件中添加新的样式规则，为body元素设置字体大小。这次使用另一种选择器——**元素选择器**，即只需要使用元素名即可。

```
body {
  font-size: 10px;
}

.thumbnail-title {
  display: block;
  margin: 0;
  padding: 4px 10px;

  background: rgb(96, 125, 139);
  color: rgb(202, 238, 255);
}
```

将body元素的font-size设置为10px。

样式表中很少会用到元素选择器，因为很少需要给文档中的某种标签全部设置为相同样式。再者，元素选择器限制了样式的复用性——使用元素选择器后，很可能到样式表的最后不得不一直重复相同的声明。如果想更改样式，维护起来也很麻烦。

不过在这里使用body作为选择器正好能满足优先级的需要。一个文档中只会有一个<body>元素，无须复用样式。

保存styles.css，在Chrome中查看页面（如图3-9所示）。

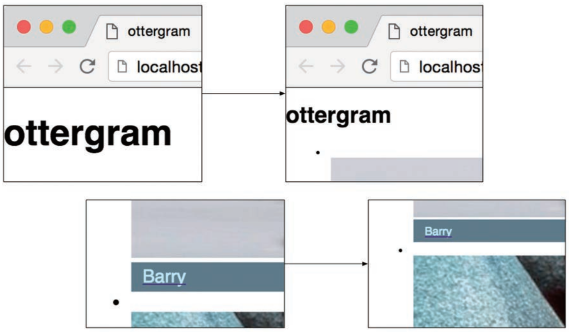


图3-9　为body设置字体大小后的效果

主标题和缩略图标题都变小了，不知这是否符合你的预期。主标题直接位于声明字体大小的body元素中，而缩略图标题却被嵌套了数层。实际上，通过样式规则，包括font-size在内的许多样式会作用于指定的元素及其**后代**元素。

文档结构可以用树形图来描述，如图3-10所示。通过树形图展示元素是实现DOM可视化的不错方式。

被某个元素所包含的元素被称为后代。这里所有的span都是body的后代（ul以及li也都是），所以它们都继承了body的font-size样式。

在开发者工具的DOM树图中找到并选中一个span元素，在Styles面板中可以看到标签为Inherited from a、Inherited from li、Inherited from ul的几块内容，这三个部分展示了从不同层级中继承的浏览器默认样式。在Inherited from body下面可以看到，在styles.css中为body设置的

`font-size`属性也被继承了（如图3-11所示）。

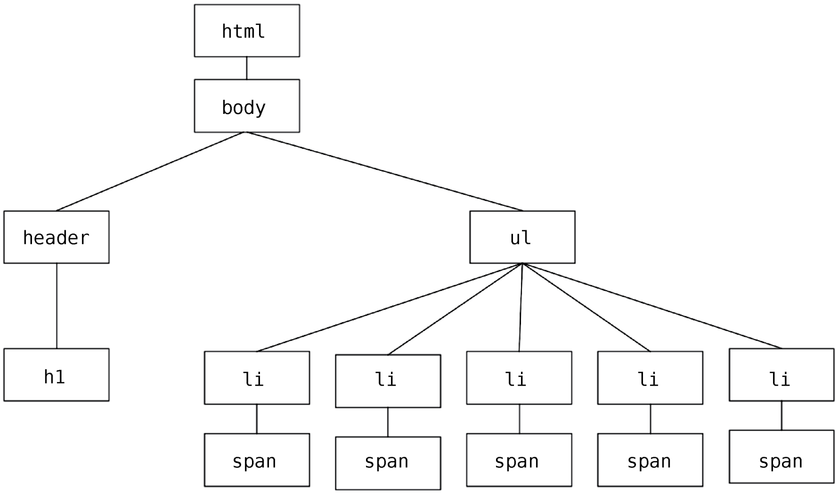


图3-10　Ottergram的树形结构

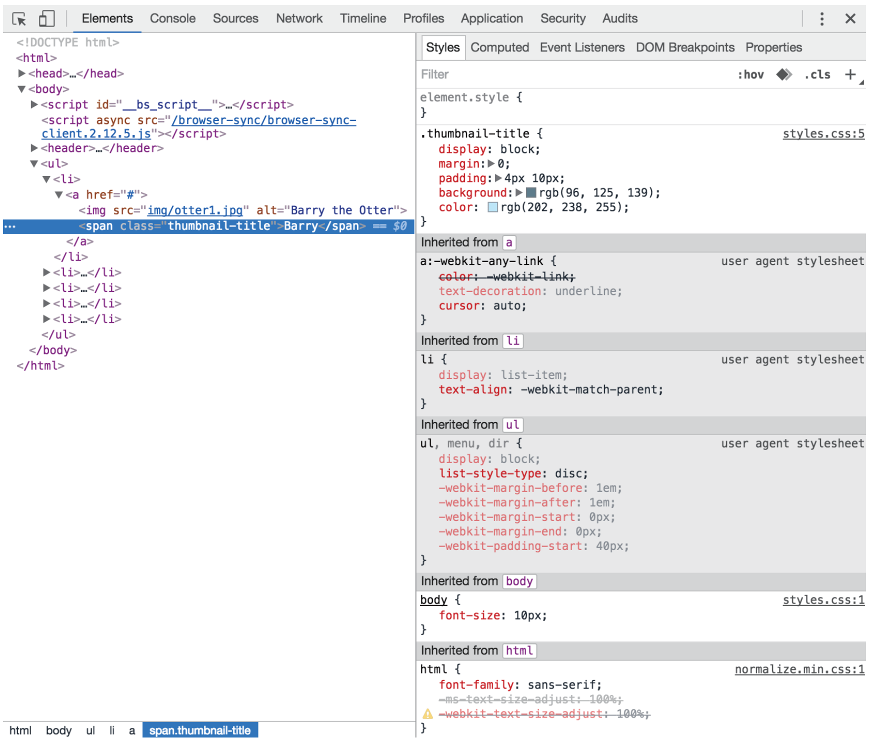


图3-11　继承祖先元素的样式

如果在另外一个层级（如`ul`中）设置字体大小，会是什么效果呢？层级更接近祖先元素，样式优先级越高。因此，在styles.css中为`ul`设置的字体大小会覆盖为`body`设置的，而为`span`设置的大小则会覆盖前两者的。

在DOM树图中点击`ul`元素可以动态调试样式。在这里添加的样式会立即反映到页面上，但不会影响实际项目文件。

在样式面板顶部，可以看到标签为`element.style`的部分。点击大括号之间的任意位置即可开始添加样式（如图3-12所示）。

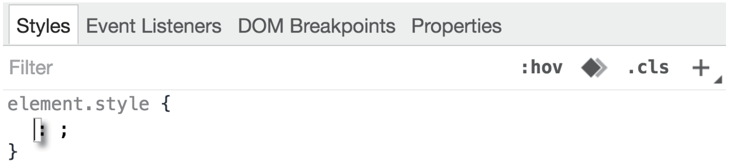


图3-12　添加样式

输入`font-size`，开发者工具会弹出提示（如图3-13所示）。

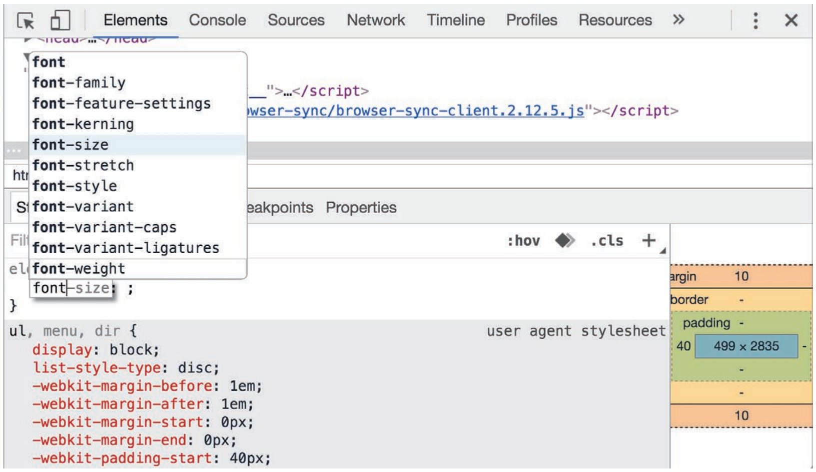


图3-13　样式面板自动补全

选择`font-size`并按下Tab键。输入一个较大的值，如50px，然后按下回车。你可能需要滚动页面，不过已经可以看到，`ul`已将`body`的`font-size`样式覆盖（如图3-14所示）。

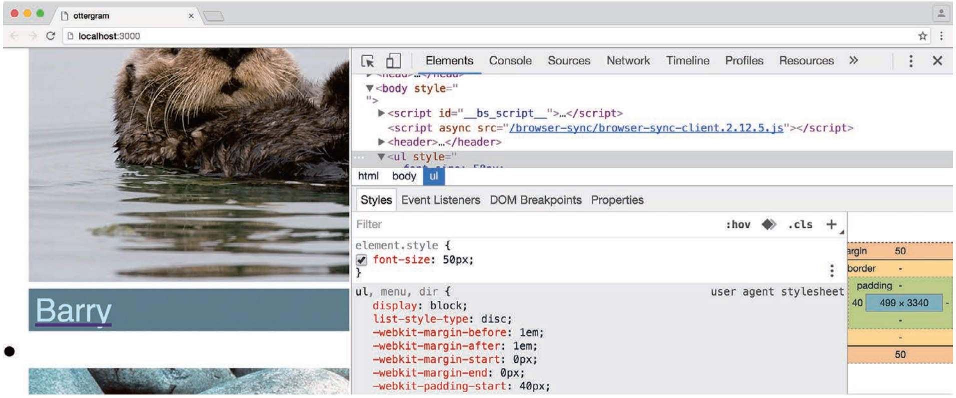


图3-14　ul字体大小设置为50px

并非所有属性都能继承，比如`border`就不行。想知道某个属性是否可以继承，请查看MDN相应的参考页面。

回到styles.css中。找到`.thumbnail-title`类，更新样式声明，使用稍大一些的字号覆盖`body`的`font-size`样式。

```
body {
  font-size: 10px;
}

.thumbnail-title {
  display: block;
  margin: 0;
  padding: 4px 10px;

  background: rgb(96, 125, 139);
  color: rgb(202, 238, 255);

  font-size: 18px;
}
```

将`.thumbnail-title`类的元素字体大小调整至18px。

保存styles.css，再去浏览器中看看效果（如图3-15所示）。

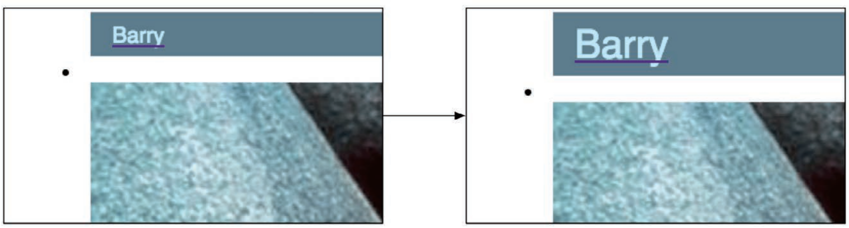


图3-15　更新样式后的缩略图标题

看起来不错，可是浏览器默认样式给.thumbnail-title类元素加上了下划线。这是因为.thumbnail-title类和.thumbnail-image类的元素都被包裹在<a>标签中，因此继承了下划线样式。

这里不需要下划线，所以要在样式文件中添加新的样式规则，修改<a>标签的text-decoration属性。这次用什么选择器呢？

如果确定要为缩略图标题和Ottergram项目中的全部其他锚元素移除下划线，只需使用元素选择器即可。


```
a {
  /* style declaration */
}
```

（/* */之间的文本是CSS**注释**，它们会被浏览器忽略。注释是开发者留下的笔记，以供日后参考。）

如果需要将锚用于其他目的（并且使用不同的样式），可以像这样添加一个**属性选择器**：

```
a[href]{
  /* style declaration */
}
```

上面的选择器会匹配所有带有href属性的锚元素。当然，锚元素通常都有href属性，所以这样可能还无法精确地匹配缩略图和缩略图标题。可以通过指定属性值，让属性选择器更加精确，如下：

```
a[href="#"]{
  /* style declaration */
}
```

这个选择器现在只会匹配href值为#的锚元素。

另外，还可以单独使用属性选择器，有无属性值均可：

```
[href]{
  /* style declaration */
}
```

根据现实情况，Ottergram项目相当简单，除了缩略图和缩略图标题外不会用到锚元素。因此使用元素选择器是安全的，也是最直观的，优先级刚刚好。

在styles.css中添加新的样式声明：

```
body {
  font-size: 10px;
}

a {
  text-decoration: none;
}

.thumbnail-title {
  ...
}
```

保存文件，查看浏览器。下划线已经消失，缩略图标题也变好看了（如图3-16所示）。



图3-16　去掉下划线后的效果

请注意，不要移除普通文本（不属于显眼的主题、标题、说明的文本）中链接的下划线。下划线是带链接的文本的重要视觉指示，符合用户心理预期。去掉这里的下划线是因为缩略图并不需要这样的视觉指示，用户理所当然地认为它们是可点击的。

后面几章会使用类名选择器为缩略图、图片无序列表、包含缩略图和缩略图标题的列表项、头部等元素添加样式。还会为index.html的`h1`、`ul`、`li`、`img`这些元素添加类名，以备不时之需。

```
...
  </head>
  <body>
    <header>
      <h1>ottergram</h1>
      <h1 class="logo-text">ottergram</h1>
    </header>
    <ul>
    <ul class="thumbnail-list">
      <li>
      <li class="thumbnail-item">
        <a href="#">
          <img src="img/otter1.jpg" alt="Barry the Otter">
          <img class="thumbnail-image" src="img/otter1.jpg" alt="Barry the Otter">
          <span class="thumbnail-title">Barry</span>
        </a>
      </li>
      <li>
      <li class="thumbnail-item">
        <a href="#">
          <img src="img/otter2.jpg" alt="Robin the Otter">
          <img class="thumbnail-image" src="img/otter2.jpg" alt="Robin the Otter">
          <span class="thumbnail-title">Robin</span>
        </a>
      </li>
      <li>
      <li class="thumbnail-item">
        <a href="#">
          <img src="img/otter3.jpg" alt="Maurice the Otter">
          <img class="thumbnail-image" src="img/otter3.jpg" alt="Maurice the Otter">
          <span class="thumbnail-title">Maurice</span>
        </a>
      </li>
      <li>
      <li class="thumbnail-item">
        <a href="#">
          <img src="img/otter4.jpg" alt="Lesley the Otter">
          <img class="thumbnail-image" src="img/otter4.jpg" alt="Lesley the Otter">
```

```
          <span class="thumbnail-title">Lesley</span>
        </a>
      </li>
      <li>
      <li class="thumbnail-item">
        <a href="#">
          <img src="img/otter5.jpg" alt="Barbara the Otter">
          <img class="thumbnail-image" src="img/otter5.jpg" alt="Barbara the Otter">
          <span class="thumbnail-title">Barbara</span>
        </a>
      </li>
    </ul>
...
```


类名添加完成后，添加样式就会方便很多。

应尽量使用类名选择器。使用描述性较强的类名能够使代码的编写和维护更加方便。此外，还可以为元素添加多个类名，提高灵活性。

记得保存index.html，然后再往后看。

3.6　图片自适应

遵循原子样式模式，接下来为图片添加样式。不过因为图片实在太大了，所以除非窗口很大，否则它们看起来是被切断的。为`.thumbnail-image`类添加样式规则，使图片适应窗口。

```
...
a {
  text-decoration: none;
}

.thumbnail-image {
  width: 100%;
}

.thumbnail-title {
  ...
}
```

将`width`属性设置为100%，将图片约束在父容器宽度之内。这样一来，当浏览器变宽时，图片也会等比例变大。保存styles.css，切换到浏览器，调整浏览器窗口大小。可以看到图片随着浏览器窗口变大缩小，但始终保持原始比例。图3-17展示了Ottergram在宽、窄窗口下的样式。

靠近细看，`.thumbnail-title`元素周围的空白不见了，标题像是与下面的图片合为一体。在styles.css中将`.thumbnail-image`的`display`属性设置为`block`。

```
...
.thumbnail-image {
  display: block;
  width: 100%;
}
...
```

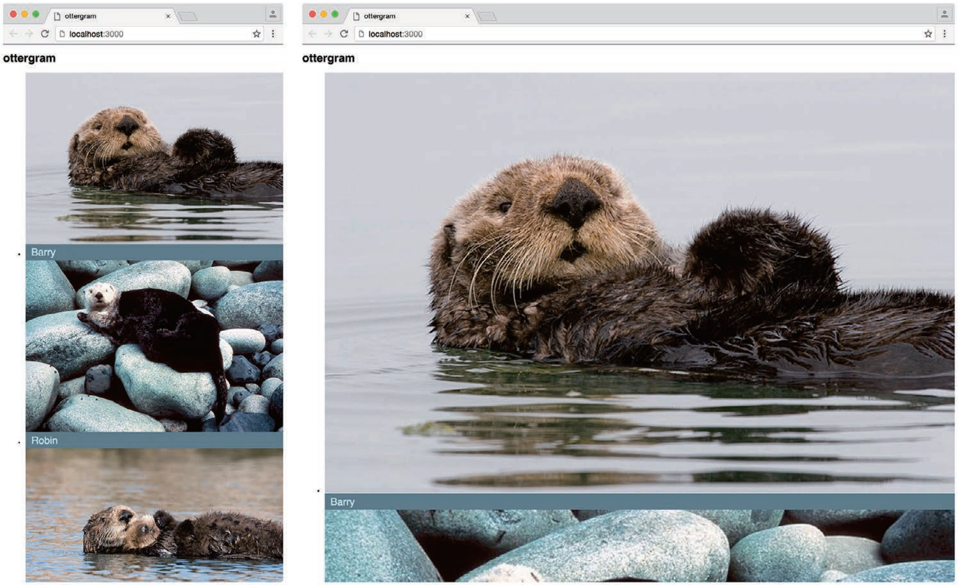


图3-17　图片宽度自适应

这样图片与标题之间的空白就消失了（如图3-18所示）。

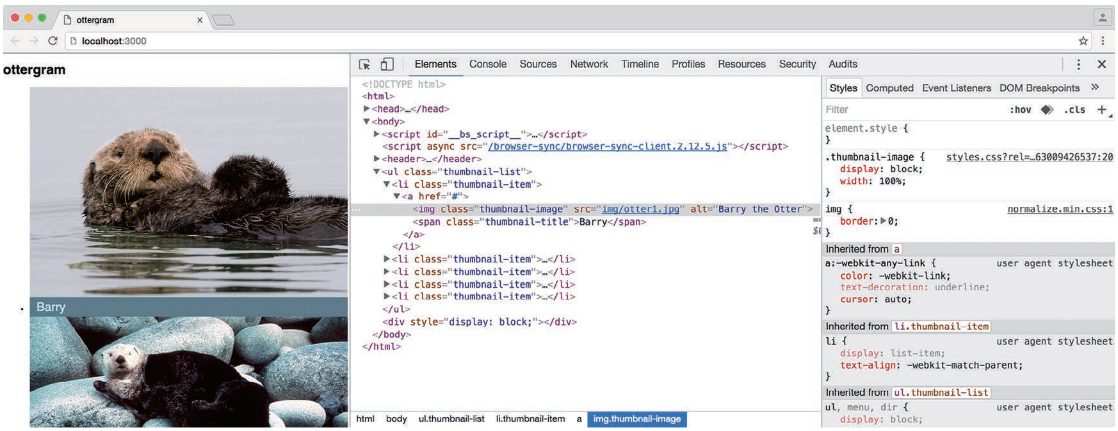


图3-18　为.thumbnail-image设置display: block

为什么这样就可以了？实际上，图片默认是display: inline的，它们的渲染规则类似于文本。渲染文本时，字母是沿着一条基线绘制的。某些字母，如p、q、y等，有一个下降部分——也就是位于基线下面的尾部。为了容纳它们，基线之下会留有一些空白。

将display属性设置为block就可移除空白，因为此处再无需要容纳的文本（以及其他任何与图片一起渲染的display: inline元素）。

3.7　颜色

该学习颜色知识了。为body元素和.thumbnail-item类添加颜色样式如下：

```
body {
  font-size: 10px;
  background: rgb(149, 194, 215);
}

a {
  text-decoration: none;
}

.thumbnail-item {
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);
}
...
```

为什么给.thumbnail-item设置了两次border呢？注意，两次声明使用了略有不同的颜色函数：rgb和rgba。rgba颜色函数接受第4个参数，表示的是透明度。不过有些浏览器并不支持rgba，因此声明两次是一种提供**回退**值的技巧。

所有浏览器在看到第1条声明（rgb）时，都会将其值注册为border属性值。当不支持rgba的浏览器看到第2条声明时，会直接将其忽略，使用第1条声明中的值。支持rgba的浏览器则会丢弃第1条声明，并使用第2条声明中的值。

（好奇为何body的背景色使用整数而.thumbnail-item的边框颜色使用百分数吗？稍后再讨论这个问题。）

保存样式文件，切换到浏览器查看效果（如图3-19所示）。

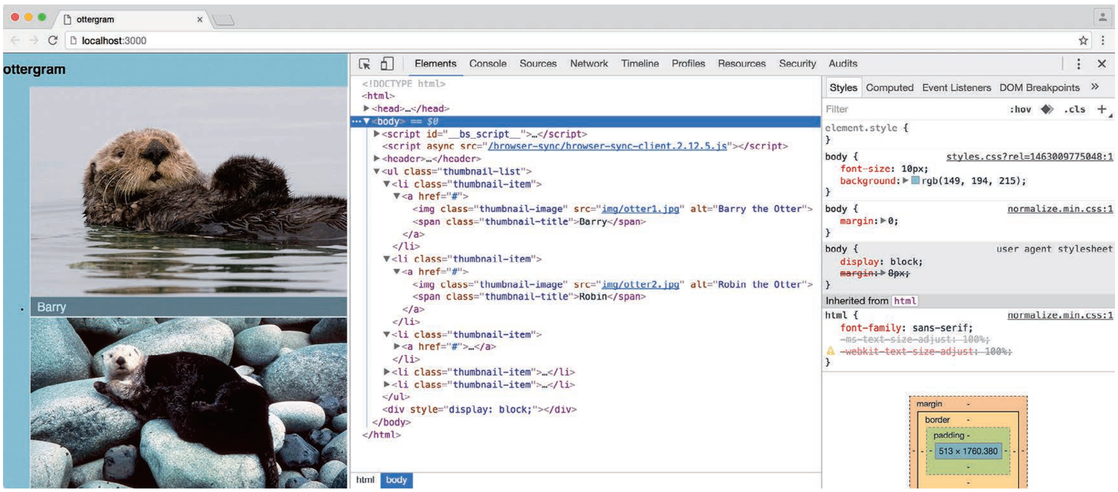


图3-19　背景色与边框

在开发者工具中，可以看到Chrome是支持rgba的。第1条声明被划去，表明没有使用rgb颜色（如图3-20所示）。

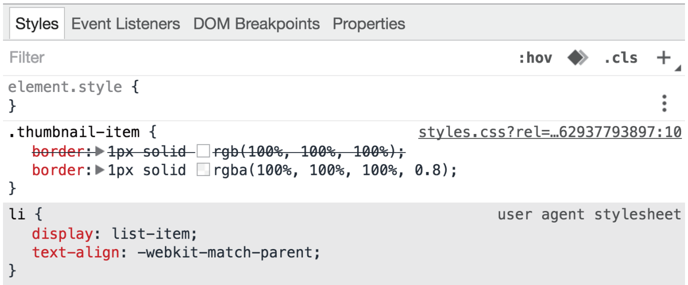


图3-20　浏览器支持rgba颜色

在开发者工具中选中body。注意在Styles面板中，背景颜色值的左侧有一个小方块展示颜色。点击方块会弹出一个颜色选择工具（如图3-21所示），可以使用该工具选择不同格式的颜色值。

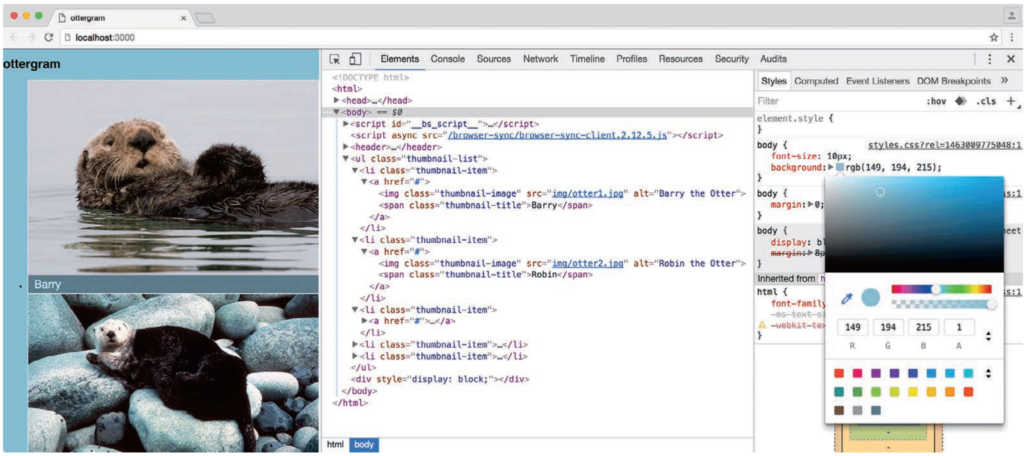


图3-21　Styles面板中的颜色选择工具

点击RGBA值右侧的上下箭头，为背景颜色切换不同的颜色格式。可以切换的格式有HSLA、HEX和RGBA。

相对于其他格式，HSLA（Hue Saturation Lightness Alpha，即色相、饱和度、明度、透明度）格式用得较少，一部分原因是多数流行的工具都没有提供精确的可用于CSS的HSLA值。如果对HSLA感兴趣，可以访问css-tricks.com/examples/HSLaExplorer。

再来看看背景颜色的HEX值：#95C2D7。HEX，即16进制（hexadecimal），是最早出现的颜色格式。每一位代表着0~15之间的值（16进制数就是将A ~ F这六个字符也当作数字），这样每一对数字就可以代表0~255之间的值。从左至右，每一对数字依次代表红、绿、蓝三种颜色的强度（如图3-22所示）。

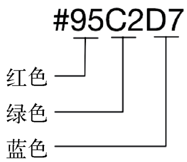


图3-22　HEX颜色值

很多人觉得HEX颜色不够直观，还有一个选择是使用RGB（Red Green Blue，红、绿、蓝）值。这种格式的每种颜色依然是0~255之间的值，但使用10进制数表示，按照颜色分开。如前所述，更先进的浏览器支持提供第4个值，指定颜色透明度，取值范围是从0.0（完全透明）到1.0（完全不透明）。透明度的正式称谓是alpha，也就是RGBA中的A。这里`body`背景颜色的RGBA值是`(149, 194, 215, 1)`。

除使用整数值外，还可以使用百分比，像前面`.thumbnail-item`的边框那样。两种方式在功能上没有差别，但不要将整数和百分比混在一起使用。

另外，Adobe提供了一个免费的在线工具帮助我们完成配色，网址是color.adobe.com。

3.8　调整空白

现在的Ottergram项目中已加入了一些漂亮的颜色，让人想起水獭们在海洋中的家。但添加颜色后，我们发现`.thumbnail-item`元素边框的内部出现了一些并不需要的空白。还有些讨厌的黑点也会影响注意力。

若想去掉这些黑点，只需将`.thumbnail-list`的`list-style`属性设置为`none`即可：

```
...
.thumbnail-item {
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);
}

.thumbnail-list {
  list-style: none;
}

.thumbnail-image {
  ...
```

去掉空白的操作和前面对`.thumbnail-image`的操作一样。每个`.thumbnail-item`都有默认空白以适应其他列表项，就像`.thumbnail-image`元素通过空白容纳相邻文本一样。为`.thumbnail-item`添加`display: block`声明，移除空白。

```
...
.thumbnail-item {
  display: block;
  border: 1px solid rgb(100%, 100%, 100%);
```

```
  border: 1px solid rgba(100%, 100%, 100%, 0.8);
}
...
```

添加这些之后，黑点和多余的空白都消失了，整个布局也更优雅，如图3-23所示。

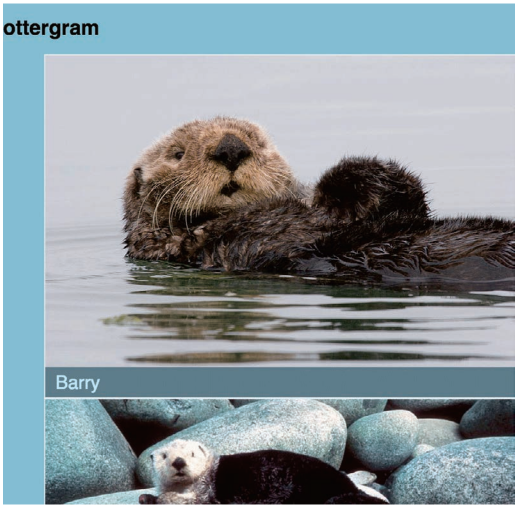


图3-23　改进后的布局

既然不需要小黑点，那为何选择带黑点的列表呢？最好按照实际功能，而非浏览器默认样式选择HTML标签。现在这里需要一个用于放置图片的无序列表，ul就是合适的选项。在第4章为项目添加大图的时候，我们会为ul容器添加样式，使其成为一个可滚动的列表。默认情况下，浏览器使用带黑点的样式展示ul，但这并非关键所在，要去掉黑点也很容易。

接下来需要为列表项之间添加空隙。目前每一项.thumbnail-item元素之间并没有任何空间，下面为相邻的缩略图添加margin。

不过，似乎并不需要为**所有**列表项添加margin。因为主标题已经有了margin，因此第一项并不需要。这也意味着无法使用.thumbnail-item类名选择器，至少无法单独使用。相反，需要用到基于元素间关系的选择器语法。

关系选择器

再看看图3-10，看起来很像家族树（family tree）吧？根据这种相似关系，产生了一系列关系选择器的名字：**后代选择器**、**子选择器**、**兄弟选择器**、**相邻兄弟选择器**。

关系选择器由两个选择器（如类名选择器或元素选择器）和一个**连接符**组成，该连接符决定了两个选择器之间的关系。为理解关系选择器如何工作，请记住浏览器是**从右向左**解析选择器的。来看几个例子。

后代选择器匹配特定类型的所有元素，它们是另一类型元素的后代元素。比方说，要选择body

元素所有的后代span元素，选择器应该这样写：

```
body span {
  /* style declarations */
}
```

这里没有用到连接符，因为选择器从右向左解析，所以可以匹配body所有的后代span元素，也就是本例中的缩略图标题。当然，也会匹配到可能添加到header或body中其他地方的span元素。

还可以在关系选择器中使用类名选择器（或属性选择器，乃至任意类型的选择器），所以前面的代码还可写成下面这样：

```
body .thumbnail-title {
  /* style declarations */
}
```

子选择器匹配特定类型的所有元素，它们是另一类型元素的直接子元素，连接符是>。若想使用子选择器匹配Ottergram中目前所有的span元素，这样写就可以了：

```
li > span {
  /* style declarations */
}
```

从右向左看，选择器匹配所有父元素为li的span元素，也就是缩略图标题。

兄弟选择器的连接符是~，它匹配拥有相同父元素的元素。但由于关系选择器的定向特质，结果与预期可能有所出入。请看下面的例子：

```
header ~ ul {
  /* style declarations */
}
```

选择器匹配所有位于**header后面**的ul元素。能够匹配到Ottergram中的ul，是因为ul前面有一个header兄弟元素。但如果将选择器倒过来（ul ~ header），就不会产生匹配结果，因为ul后面没有任何header元素。

最后一种关系选择器是相邻兄弟选择器，匹配**紧邻**指定类型的元素之后的元素，其连接符为+：

```
li + li {
  /* style declarations */
}
```

上面的选择器会选中所有紧邻li元素之后的li元素。结果为样式会作用于第2～5个li元素，因第1个li元素的前面不存在其他li元素，所以不会对其产生作用。（需要留意的是，因目前Ottergram项目结构相对简单，普通兄弟选择器和相邻兄弟选择器的效果是完全相同的。）

回到手上的任务：为除第1个列表项外的所有列表项添加顶部margin。假若使用后代选择器或子选择器匹配类.thumbnail-item或元素span、li，则样式会作用于全部5张缩略图。因此使用相邻兄弟选择器，为一张缩略图之后紧邻的另一张缩略图添加margin。

```
...
a {
  text-decoration: none;
}

.thumbnail-item + .thumbnail-item {
  margin-top: 10px;
}

.thumbnail-item {
  ...
```

保存文件，并在浏览器中查看结果（如图3-24所示）。

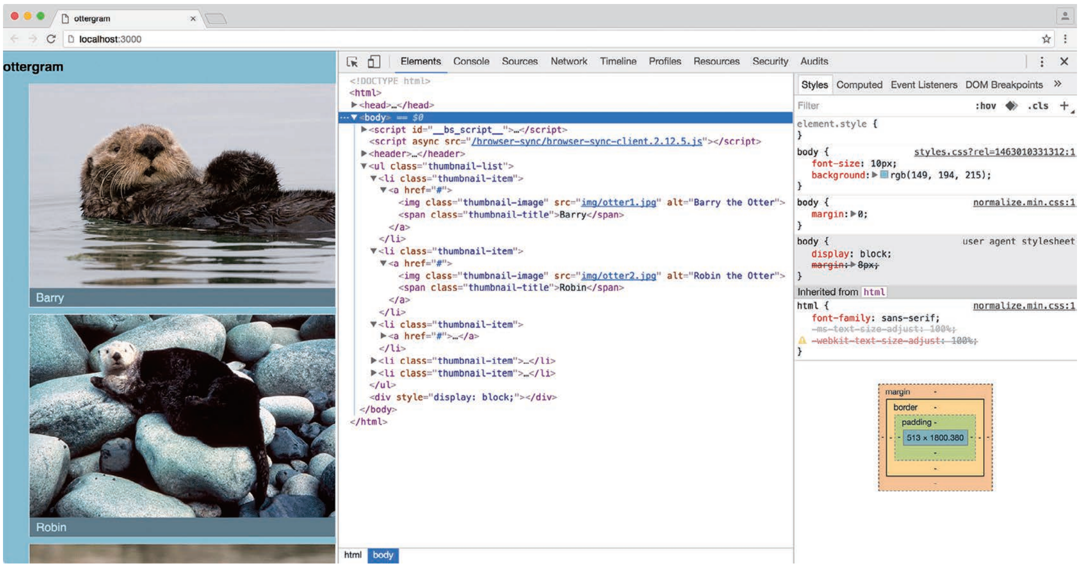


图3-24　相邻.thumbnail-item元素之间的空隙

细心的读者可能注意到了，开发者工具提供了一种简单的方式查看元素嵌套路径，这可以帮助我们编写关系选择器。点击某个li元素中的span，在Elements面板底部就能看到元素路径（如图3-25所示）。

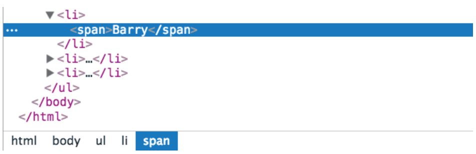


图3-25　在Elements面板中查看元素嵌套路径

改变缩略图列表外观的最后一步是打开styles.css，覆盖ul从浏览器默认样式继承来的padding，让图片不再缩进。

```
...
.thumbnail-list {
  list-style: none;
  padding: 0;
}
...
```

依然还是保存文件，并到浏览器中查看效果（如图3-26所示）。

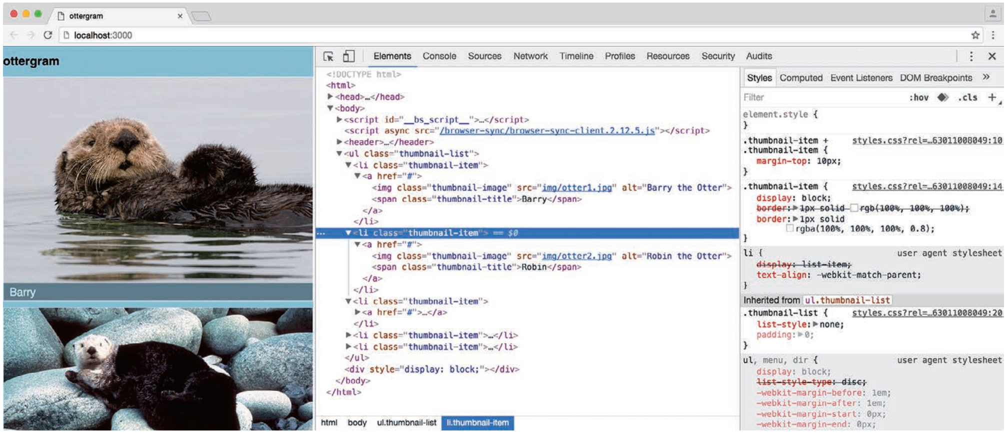


图3-26　去掉padding后的ul

Ottergram开始变漂亮了。再为头部添加一点样式，一个很棒的静态网页就出炉了。

3.9　添加字体

前面为h1元素添加了.logo-text类名。将这个类名用作选择器，在styles.css中添加一条新的样式规则，将代码插入到锚标签样式之后。（通常来说，样式顺序不同只会在同一选择器存在多条规则的情况下产生影响。在Ottergram中，样式顺序大致按照元素在代码中出现的顺序进行组织。这取决于个人偏好，可以根据自身情况自行决定。）

```
...
a {
  text-decoration: none;
}

.logo-text {
  background: white;

  text-align: center;
  text-transform: uppercase;
  font-size: 37px;
}


.thumbnail-item + .thumbnail-item {
  ...
```

首先为头部添加白色背景，然后使`.logo-text`元素中的文本居中显示，再通过`text-transform`属性将文本格式调整为大写，最后设置字体大小。图3-27展示了修改后的结果。

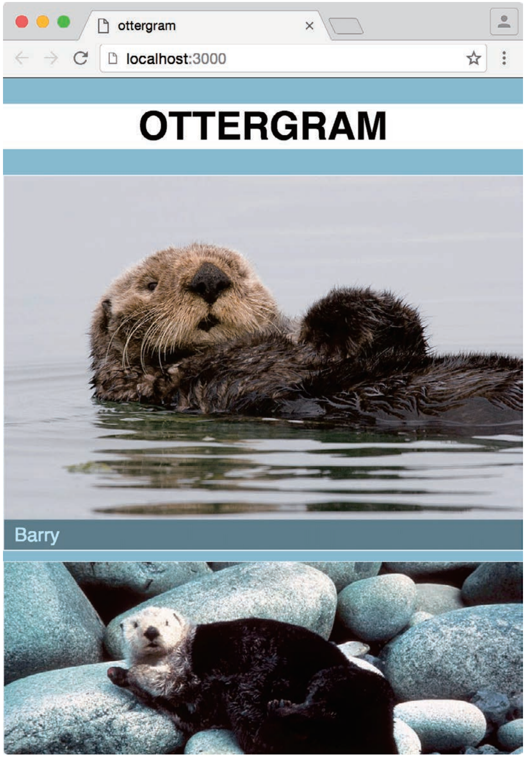


图3-27　为头部添加样式

看起来还不错，但仅仅只是不错。对可爱的水獭们来说，这么一个网站显得有些普通。为了让网页更酷一些，可以为头部设置一种不是浏览器提供的默认样式的字体。

之前下载的资源文件中已经有一些字体文件。要使用它们，请将fonts文件夹复制到项目目录下，放在stylesheets文件夹中（如图3-28所示）。

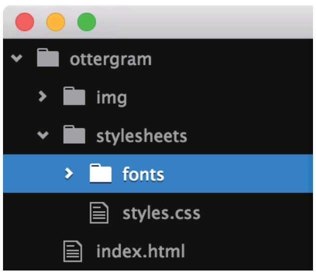


图3-28　将fonts文件夹复制到stylesheets文件夹中

现在只需要一些引用这些字体的样式即可。

资源文件中包含了每种字体的多种不同样式，不同浏览器厂商支持的字体类型也不一样。要想最大程度上支持各类浏览器，需要将所有格式都放到项目中。没错，所有格式。

通过@font-face语法可以为字体添加自定义名称，并在其他地方使用。

@font-face语句块与此前使用的语句块有些许不同，它主要包括3部分。

- 首先是font-family 属性，其值是用于标记自定义字体名称的字符串，在整个CSS文件中都能使用。
- 其次是一些src声明，指定不同的字体文件。（请注意，顺序非常重要！）
- 最后是修改字体样式的声明，如font-weight和font-style等。

在styles.css文件顶部为lakeshore字体添加@font-face声明，并在.logo-text类中添加使用新字体的样式声明。

```
@font-face {
    font-family: 'lakeshore';
    src: url('fonts/LAKESHOR-webfont.eot');
    src: url('fonts/LAKESHOR-webfont.eot?#iefix') format('embedded-opentype'),
         url('fonts/LAKESHOR-webfont.woff') format('woff'),
         url('fonts/LAKESHOR-webfont.ttf') format('truetype'),
         url('fonts/LAKESHOR-webfont.svg#lakeshore') format('svg');
    font-weight: normal;
    font-style: normal;
}

body {
  font-size: 10px;
  background: rgb(149, 194, 215);
}

a {
  text-decoration: none;
}

.logo-text {
 background: white;

 text-align: center;
 text-transform: uppercase;
 font-family: lakeshore;
 font-size: 37px;
}
...
```

不得不承认，将@font-face声明写正确的确有些麻烦，因为每个url值的顺序很重要。将这段声明复制下来作为使用时的参考倒是个不错的主意。还可以通过flight-manual.atom.io/using-atom/sections/snippets查看Atom snippets文档，学习如何创建自己的代码片段或模板。

声明自定义@font-face后，在CSS其余部分中的font-family属性中就可以使用lakeshore这个值。请在样式声明中使用font-family: lakeshore为.logo-text设置新字体。

保存styles.css，切换到Chrome，欣赏这和水獭一样酷的网页吧（如图3-29所示）。



图3-29 为头部添加自定义字体

本章完成了大量样式工作，Ottergram变得棒多了！下一章会添加一些交互，网页会变得更棒。

3.10 初级挑战：更改颜色

修改body的背景颜色，试着用开发者工具中的颜色选择器（如图3-21所示）选择一种颜色。

访问color.adobe.com可以使用更精致的调色板，为body和.thumbnail-title创建属于你的背景色彩组合。

3.11 延展阅读：优先级！当选择器发生冲突了……

前面已经讲解过如何覆盖样式了，例如在styles.css前引入normalize.css。normalize.css中的样式会成为浏览器基准，而自定义样式又优先于基准样式。

这是浏览器如何为页面中的元素选择样式的最基本概念，在前端开发中通常被称为**就近原则**：当浏览器处理CSS规则时，可能会覆盖之前处理过的规则。通过改变<link>标签的顺序即可控制浏览器处理CSS的顺序。

当样式规则选择器相同时（比如CSS和normalize.css为body元素添加了不同的margin），就相当简单了，浏览器会选择最近的样式声明。那如果有多个选择器匹配同一个元素又会怎样呢？

假设在Ottergram中有这样两条CSS规则：

```
.thumbnail-item {
  background: blue;
  }

li {
  background: red;
}
```

两种方式都匹配<li>元素，那么<li>元素会是什么颜色？虽然li { background: red; }在后面，但起作用的还是.thumbnail-item { background: blue; }。为什么呢？因为类名选择器比元素选择器优先级高（也就是说权重更高）。

类名选择器和属性选择器的优先级相同，都高于元素选择器。优先级最高的是**ID选择器**，目前尚未介绍。如果为某个元素添加了id属性，则可使用优先级最高的ID选择器。

ID属性与其他属性很相似，示例如下：

```
<li class="thumbnail-item" id="barry-otter">
```

ID选择器是在ID前面加上#：

```
.thumbnail-item {
  background: blue;
}

#barry-otter {
  background: green;
}

li {
  background: red;
}
```

上面例子中的3个选择器均可匹配<li>元素，但因ID选择器优先级最高，所以<li>背景为绿色。又因为选择器优先级各不相同，所以规则顺序不会影响结果。

请记住一点：最好不用ID选择器。在文档中，ID值是唯一的，所以不能再为其他元素设置id="barry-otter"。尽管ID选择器优先级最高，但与其相关的样式难以复用，进而成为代码维护中的“最糟实践”。

要了解更多关于优先级的内容，请访问MDN：developer.mozilla.org/en-US/docs/Web/CSS/Specificity。

specificity.keegan.st的优先级计算器是一个计算不同选择器优先级的便利工具。打开看看，了解选择器优先级到底是怎么计算的。

第 4 章

flexbox响应式布局

前端开发者的任务之一就是为使用不同设备和浏览器的用户提供最佳体验。

但这并非从一开始就是业界共识，浏览器厂商对此负有部分不可推卸的责任。在Web发展早期，各厂商之间便展开大战，竞相推出不标准的新特性，试图超越对手。为此，Web开发者得想出各种对策，检测请求文档的是哪种浏览器，使用的屏幕分辨率是多少。然后基于这些信息，为不同浏览器提供不同版本的文档。

可悲的是，要想针对特定分辨率下特定版本的浏览器，就需要为网站的每个页面创建一个特定的版本。这使得前端开发负重不堪，维护起来耗时耗力，令人沮丧。

谢天谢地，浏览器大战终于结束了，各大厂商正努力遵循同一套规范——现代前端开发者们终于可以安心地只编写一套代码，为不同版本的浏览器提供不同页面的日子一去不返。然而，这并不意味着不再需要为不同尺寸、方向的显示器定制页面。一些新技术，如本章将要学习的flexbox，允许根据用户屏幕尺寸进行布局调整，而无须重复的文档。

本章会将Ottergram由简单的图片列表拓展为可交互用户界面。通过flexbox和CSS定位，可以创建一组界面组件，它们根据浏览器窗口尺寸进行调整，并保持总体布局不变。等到本章结束，Ottergram会包含一个缩略图滚动列表和一个单张大图展示区域（如图4-1所示）。

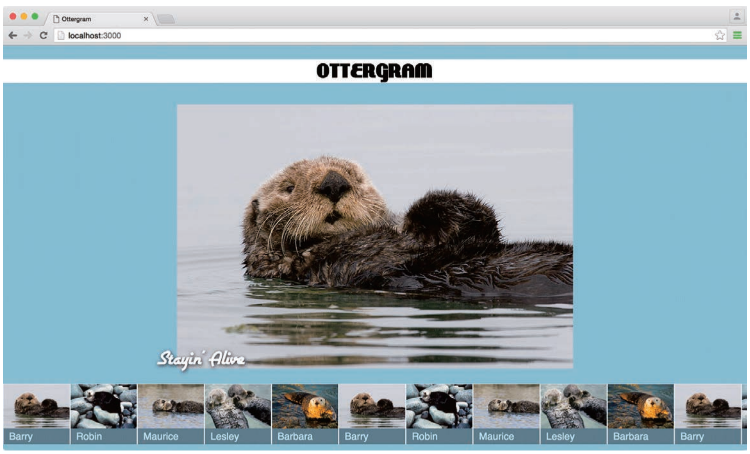


图4-1　flex布局的Ottergram

工作分两步进行：首先添加一些必要的标记和样式，用于展示大图并使缩略图变小且可滚动；接着添加一些样式，使窗口部分能够伸缩，以适应不同尺寸的屏幕和窗口。

4.1　界面拓展

自从iPhone面世，使用智能手机替代台式机、笔记本访问网络的趋势节节高升。

对前端开发者而言，这种趋势证实了**移动优先**是最佳的设计方法：先为小屏幕进行设计，接着是手持设备屏幕，最后是台式机屏幕。

Ottergram的简单布局已是移动端友好型，适合在较小的屏幕上显示文本和图像，因此可以直接进行下一步布局。

采用垂直滚动列表虽未尝不可，但如果还能让用户看到大图会更好。Ottergram计划在展示大图的同时，使缩略图列表水平滚动。目前先将大图放在列表下面，如图4-2所示。

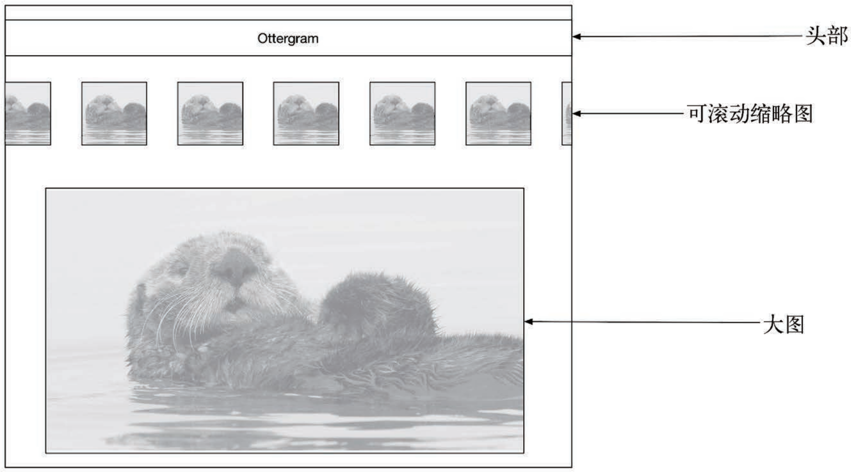


图4-2　Ottergram的新布局

先从添加大图开始吧。

4.1.1　添加大图

目前先将大图设置为固定的一张图。在第6章中将会添加功能，使我们在点击缩略图时能够切换大图。

在index.html中添加一段代码用来展示大图：

```
...
        <li class="thumbnail-item">
          <a href="#">
            <img class="thumbnail-image" src="img/otter5.jpg" alt="Barbara the Otter">
            <span class="thumbnail-title">Barbara</span>
```

```
        </a>
      </li>
    </ul>

    <div class="detail-image-container">
      <img class="detail-image" src="img/otter1.jpg" alt="">
      <span class="detail-image-title">Stayin' Alive</span>
    </div>

  </body>
</html>
```

添加了一个类名为detail-image-container的<div>。<div>是常用的内容容器，通常用于为一段内容添加样式。

在<div>中添加了用于展示大图的<img>，以及一个包裹着大图标题的<span>。分别为<img>和<span>添加类名detail-image和detail-image-title。

保存index.html，切换到styles.css，并在最后添加样式，限制.detail-image类的宽度。

```
...
.thumbnail-title {
  ...
}

.detail-image {
  width: 90%;
}
```

保存styles.css，启动browser-sync，在Chrome中打开项目（如图4-3所示）。（命令是browser-sync start --server --browser "Google Chrome" --files "stylesheets/*.css, *.html"。）

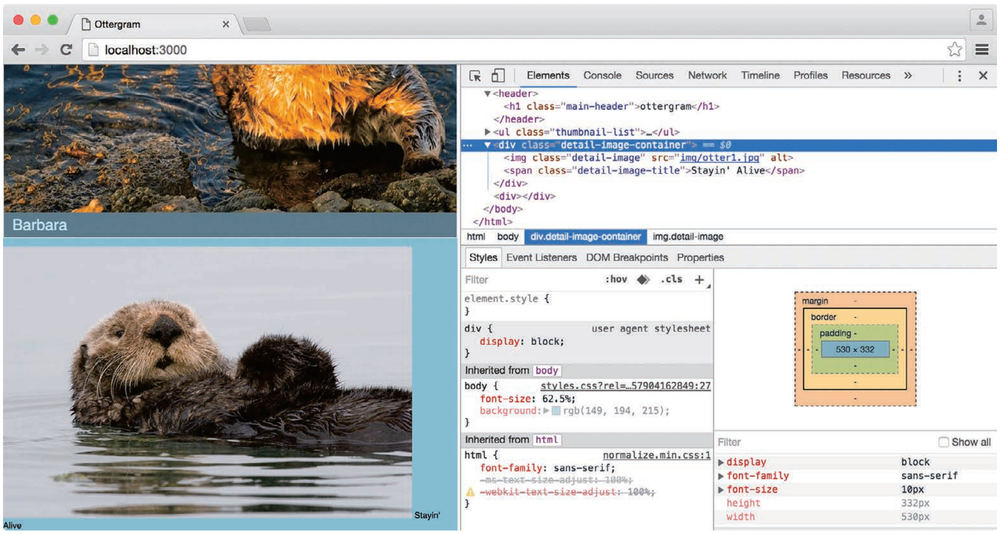


图4-3　大图初始样式

.detail-image出现在页面底部，比缩略图略窄一些。大图宽度被设置为其容器宽度的90%，右侧还有一些空余。浏览器将.detail-image-title中的文本放到了空余部分中。（稍后会为文本添加样式。）

试着改变浏览器窗口大小，你会发现一个问题：当调整缩略图的宽度时，大图可能会被挤出视图。这个问题稍后再解决。

4.1.2　缩略图水平布局

接着来更新.thumbnail-list和.thumbnail-item类的样式，使图片水平滚动。

将index.html中的5个
复制一遍，生成足够多的内容以测试滚动功能。选中<ul class="thumbnail-list">和
两行之间的内容，复制并粘贴到
之前。最终一共有10项，为otter1.jpg到otter5.jpg这5张图各重复一次。

记得保存index.html。在开发过程中复制内容是一个不错的办法，可用于模拟更加健壮的项目，还能预览代码如何处理真实场景。

为使缩略图水平轮动，必须限定缩略图列表中每一项的宽度，并使其水平排成一行。

此前多次使用过的display: block这次无法派上用场了，因为它会在元素前后产生换行效果。不过另一个相关的属性display: inline-block倒正好适用。使用display: inline-block时，元素盒子会如display: block一般展现，但不会产生换行——这样一来，缩略图就可以排成一行。

在styles.css中修改.thumbnail-item类的display声明，并添加width声明：

```
...
.thumbnail-item {
  display: block;
  display: inline-block;
  width: 120px;
  border: 1px solid rgb(100%, 100%, 100%, 0.8);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);
}
...
```

（注意，Atom中的linter可能会警告“同时使用width和border可能导致元素大小超出预期”。这是因为width只针对元素盒子的内容部分，而不是内边距或边框部分。无须理会这个警告。）

如果将.thumbnail-item元素的宽度设置为120px，.thumbnail-image同样也会固定下来，因为.thumbnail-image自适应为其容器的宽度。

那么为何不为.thumbnail-image设置width: 120px呢？我们需要让.thumbnail-image和.thumbnail-title宽度一致，但没有分别为两者设置宽度，而是为它们共同的父元素设置宽度。如此一来，若要改变width，只需修改一处。一般来说，让内部元素根据容器自适应是一种不错的实践。

保存styles.css，在Chrome中查看页面。可以看到.thumbnail-item元素依次排成一排——不过在容器宽度被填满后产生了换行（如图4-4所示）。

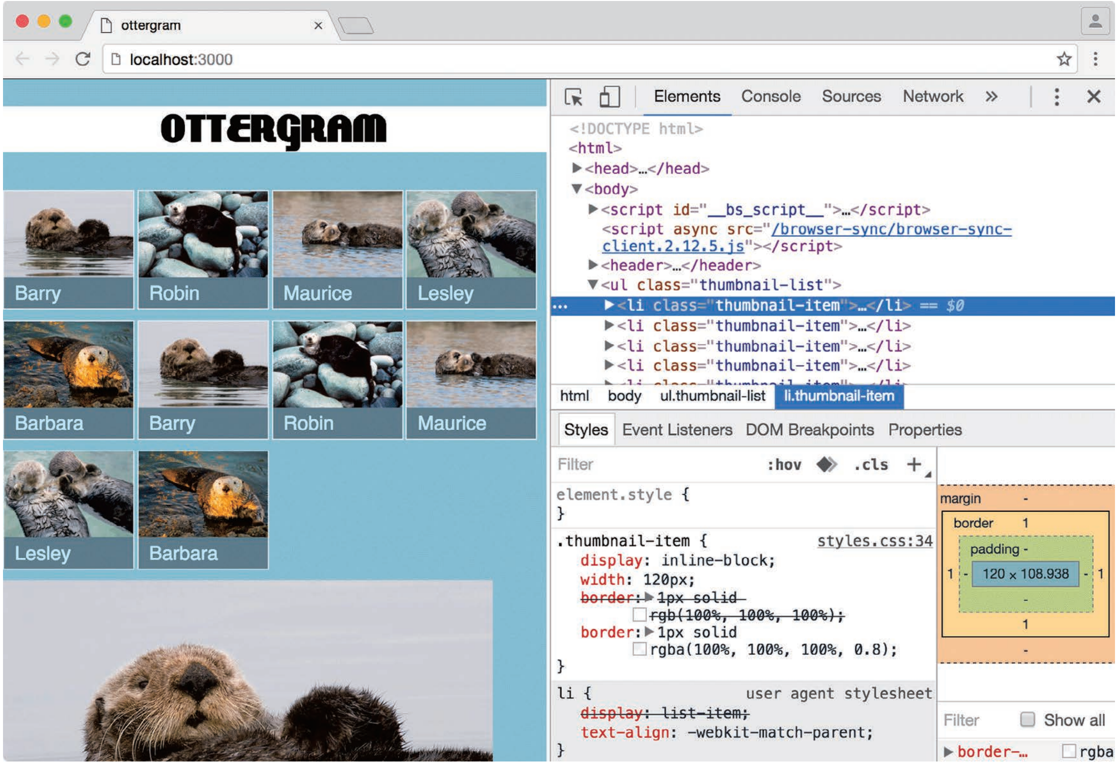


图4-4　inline-block产生换行

为得到预期的滚动效果，需要在`.thumbnail-list`中设置禁止换行并允许滚动。

```
...
.thumbnail-list {
  list-style: none;
  padding: 0;

  white-space: nowrap;
  overflow-x: auto;
}
...
```

`white-space: nowrap`声明禁止`.thumbnail-item`元素换行。`overflow-x: auto`则告诉浏览器，在`.thumbnail-list`元素的水平方向（X轴）上添加滚动条，以容纳超出的部分。若没有此声明，则需要滚动整个页面方能看到超出部分。

再次保存文件，在浏览器中看下效果。这下缩略图都在一排了，还能水平滚动（如图4-5所示）。

对于改进Ottergram界面来说，这已经是一个不错的开始——但还算不上完美，因为页面还不能很好地适应很多其他尺寸的屏幕，尤其是较当前所用屏幕而言大很多或小很多的屏幕。

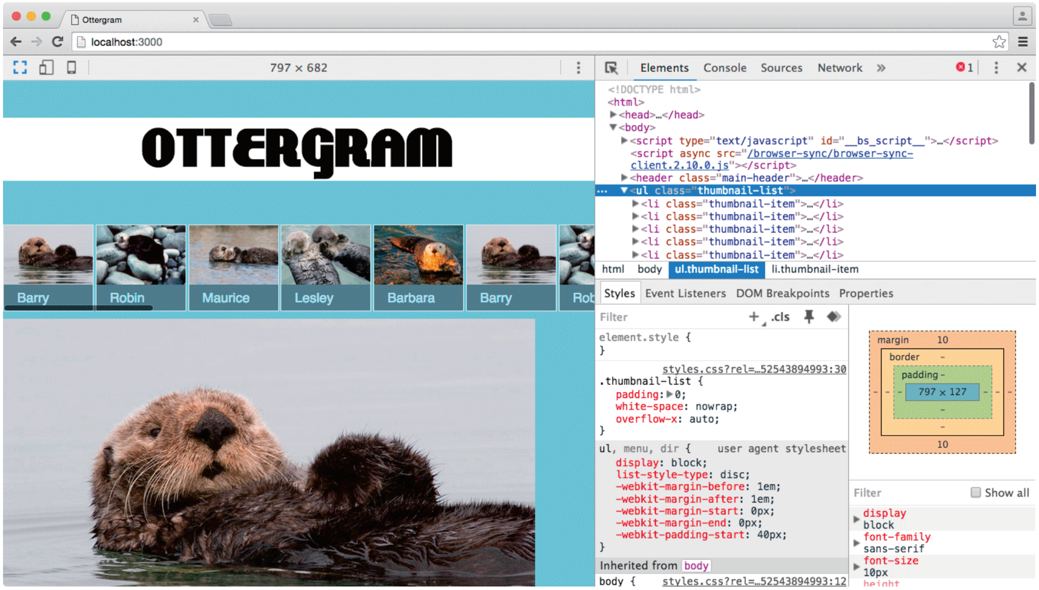


图4-5 可以水平滚动的缩略图

接下来的两节会添加一些代码，为Ottergram添加更加流体化的布局，并允许UI在不同布局间切换，以适应不同尺寸的屏幕。

4.2 flexbox

前面已经见过将`display`样式指定为`block`和`inline`的情况。像刚刚完成的滚动列表中的缩略图项这样的行内元素（inline element）会逐一相邻排列，而块级元素（block element）则占据整个水平方向。

还有另外一种思考方式，即块级元素从上向下**流动**，而行内元素从左向右**流动**（如图4-6所示）。

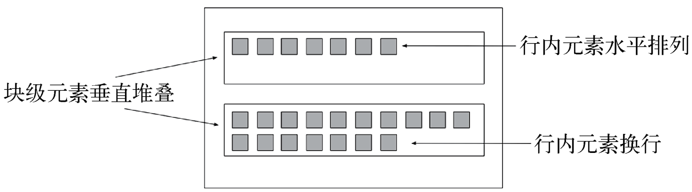


图4-6 块级元素与行内元素

`display`属性告诉浏览器元素在页面布局中如何排列。对博客或在线百科这类网站来说，`inline`和`block`值已经足够使用。但针对基于Web的邮件、社交媒体等应用型布局的站点，可以根据新的CSS规范对其采用更加动态的布局——也就是**弹性盒布局**，又称为flexbox。

flexbox CSS属性可以确保缩略图和大图区域填满整个屏幕，并保持彼此间的相对比例——这

正是Ottergram所需要的。此外，flexbox属性还能够将大图区域的内容水平、垂直居中，但如果使用标准盒模型则很难实现。

4.2.1　创建flex容器

在添加第一个flexbox属性前，先将`<html>`和`<body>`的高度设置为100%。`<html>`元素是DOM树的根元素，`<body>`是其子元素。将这两者的高度设置为100%，允许内容填满浏览器或设备窗口。

```
@font-face {
  ...
}

html, body {
  height: 100%;
}

body {
  font-size: 10px;
  background: rgb(149, 194, 215);
}
...
```

注意上面代码中由逗号分隔的两个选择器。任意类型的选择器都可以如此组合，以设置一些共同样式。

此外还可注意到，`body`选择器现在已经有两条样式规则。当浏览器看到附加的样式声明时，会将其添加到该选择器现有的样式信息中。如本例的浏览器首先看到`<body>`的`height`应该为100%，并将信息存储下来。然后继续读下一条规则，将`background`和`font-size`信息和`height`存在一起。

现在可以开始创建第一个**flex容器**了。flex容器能够控制其子元素（**flex项目**）的布局。在flex容器中，flex项目的大小和位置沿着**主轴**和**侧轴**出现（如图4-7所示）。

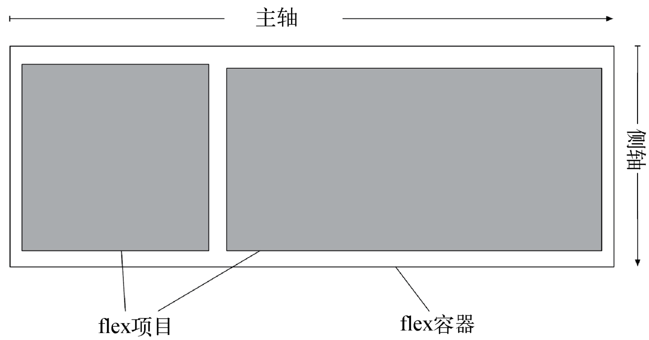


图4-7　flex容器的主轴和侧轴

为<body>添加display: flex声明，将其转变为flex容器。

```
...
body {
  display: flex;

  font-size: 10px;
  background: rgb(149, 194, 215);
}
...
```

如果立马保存代码，浏览器中的样式会很悲催，如图4-8所示。这是因为主轴从左向右将所有的flex项目（<body>的所有子元素）排成了一行。

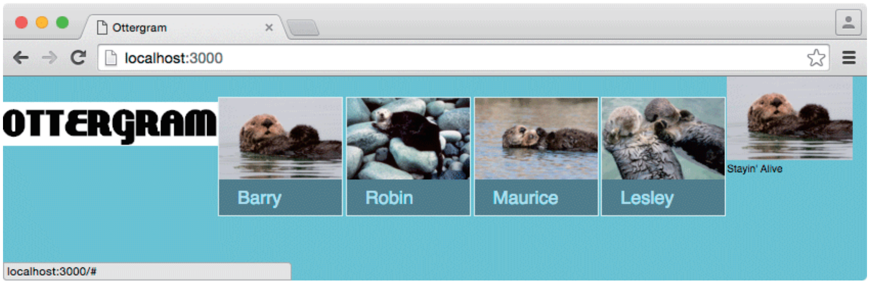


图4-8 沿主轴排列的flex项目

不过我们也看到flex项目为适应空间有所收缩，而不是换行，这倒是一个好消息。第二个好消息是只需要一行样式就能解决布局问题（差不多就一行）。

4.2.2 改变flex-direction

为<body>元素设置flex-direction便可以解决布局问题：

```
...
body {
  display: flex;
  flex-direction: column;

  font-size: 10px;
  background: rgb(149, 194, 215);
}
...
```

这样就对调了flex容器的主轴和侧轴，如图4-9所示。

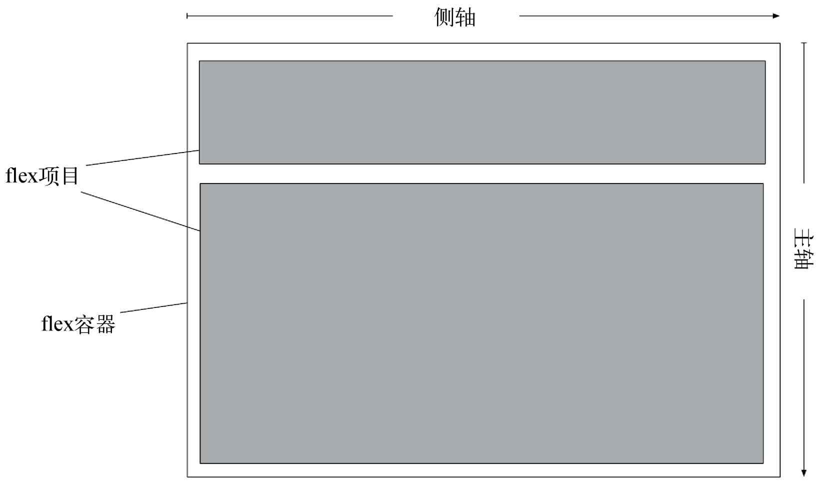


图4-9　设置`flex-direction: column`之后的主轴和侧轴

将`flex-direction`设置为`column`后，Ottergram基本恢复正常。但是当浏览器窗口的宽度远大于高度时，会出现如图4-10所示的视觉问题。

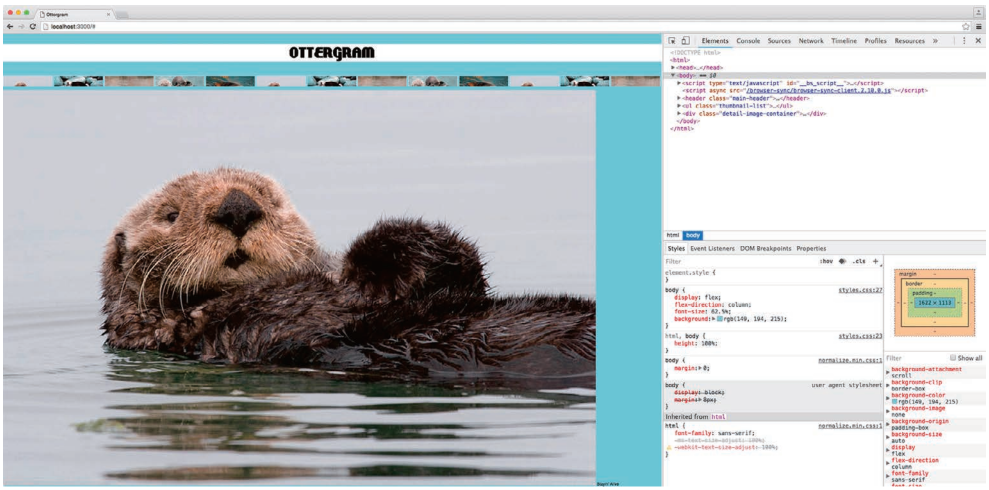


图4-10　页面变宽时缩略图几近消失

只要新增一个包裹元素，添加一些新的flexbox属性，就能补救问题。

4.2.3　flex项目中的元素分组

`<body>`元素有三个flex项目：`<header>`、`.thumbnail-list`和`.detail-image-container`容器。在Ottergram的开发（以及使用）过程中无论发生什么，`<header>`在布局和复杂度方面基

本不会有太多变动。它位于页面顶部，展示文本，仅此而已。

另一方面，`.thumbnail-list`和`.detail-image-container`及其内容在布局和复杂度上则很可能发生变化。任何一个发生变化，很可能会影响到另一个。

基于前述原因，我们将`.thumbnail-list`和`.detail-image-container`分到一个组，将它们放在一个flex容器中。使用类名为`main-content`的`<main>`标签将它们包裹起来（如图4-11所示）。

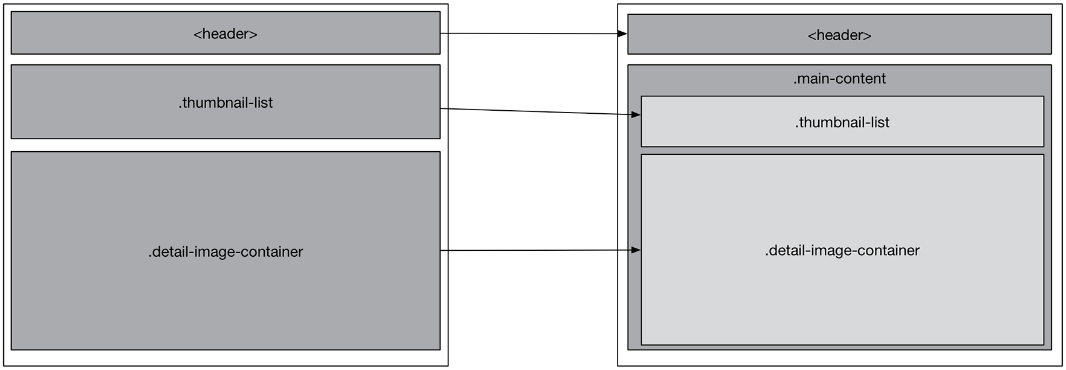


图4-11　包裹`.thumbnail-list`和`.detail-image-container`

在index.html中做以下更改：为`<header>`添加`main-header`类名，并将`.thumbnail-list`（`<ul>`）和`.detail-image-container`（`<div>`）放在类名为`main-content`的`<main>`元素中。

```
...
  <body>
    <header>
    <header class="main-header">
      <h1 class="logo-text">ottergram</h1>
    </header>
    <main class="main-content">
      <ul class="thumbnail-list">
      ...
      </ul>

      <div class="detail-image-container">
        <img class="detail-image" src="img/otter1.jpg" alt="">
        <span class="detail-image-title">Stayin' Alive</span>
      </div>
    </main>
...
```

现在`.main-header`和`.main-content`是`<body>`的两个flex项目了。

将`.thumbnail-list`和`.detail-image-container`放在`.main-content`中之后，便可以随意指定`<header>`的高度，`<body>`在垂直方向上余下的空间就是`.main-content`所占据的空间。这样就可以在不影响头部的情况下，为`.thumbnail-list`和`.detail-image-container`分配空间。

保存index.html。`body`中只剩下两个flex项目，可以使用`flex`属性指定它们的相对尺寸。

4.2.4　flex缩写属性

flex容器将其空间分配给内部的flex项目。若没有在主轴方向上给flex项目指定尺寸，默认情况下，flex容器会根据flex项目数量平均分配空间，在主轴方向上的每个项目获得相同的空间比例，如图4-12所示。

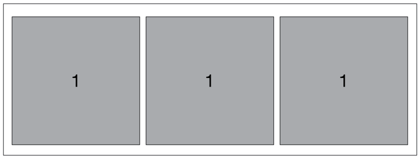


图4-12　3个flex项目平均分配空间

假设图4-12中的某个flex项目较其他项目来说略显贪婪，想要索取两份空间。这种情况下，flex容器会将主轴方向的空间分为4等份，那个贪婪的flex项目独占其中2份（也就是一半），余下两个项目各得1份（如图4-13所示）。

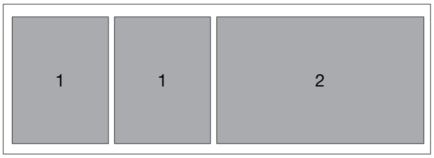


图4-13　3个flex项目不平均分配空间

Ottergram中的`.main-content`是这个贪婪的元素，它要在主轴方向上尽可能占有更多空间；而`.main-header`则尽量少占空间。

`flex`属性为flex项目指定了能占用的空间。这是一个缩写属性，如图4-14所示。

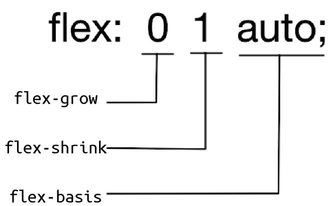


图4-14　`flex`缩写属性及其值

强烈建议直接使用`flex`替代具体属性，这能够避免因无意间遗漏某个属性而造成的不符合预期的结果。

第一个值是我们当下要关注的，它决定了flex项目的拉伸程度。默认情况下，flex项目不拉伸。对`.main-header`来说，默认行为恰好合适，但是`.main-content`则需要拉伸。

在styles.css中为`.main-header`添加样式，指定`flex`为默认值`0 1 auto`。

```
...
a {
  text-decoration: none;
}

.main-header {
  flex: 0 1 auto;
}

.logo-text {
  background: white;
  ...
```

值`0 1 auto`可解读为“无须进行拉伸，如有必要将会收缩，自动计算大小”。最终结果就是`.main-header`只会占据它所需的空间，一点不多。

接着为`.main-content`添加样式声明，将其`flex`属性设置为`1 1 auto`。

```
...
.logo-text {
  ...
}

.main-content {
  flex: 1 1 auto;
}

.thumbnail-item + .thumbnail-item {
  ...
```

`.main-content`的`flex`声明中的第一个值对应`flex-grow`属性，值为1意味着会尽可能拉伸。`.main-content`唯一的兄弟元素`.main-header`已在前面声明不会进行拉伸，所以`.main-content`会拉伸并占据剩余的全部空间。

`.main-header`和`.main-content`两个元素作为`<body>`的flex项目，根据各自的需要占据`<body>`的弹性空间。接下来对`.main-content`元素的布局进行调整。

4.2.5 flex项目的排序与对齐方式

flexbox技术允许将flex项目进一步细分为flex容器，这允许我们专注于某一层的布局。下面很快会将`.main-content`变成flex容器。

使用嵌套的flex容器时，从最小、最内部的元素开始逐渐向上创建样式的方法已不再适用，而从最外层元素开始一路向下会更有用。

接下来会将`.main-content`转换为主轴为垂直方向的flex容器，并为其flex项目指定`flex`属性，使`.thumbnail-list`占据默认大小的空间，`.detail-image-container`根据剩余空间进行拉伸。最后，将`.thumbnail-list`移动到`.detail-image-container`下面（如图4-15所示）。

在styles.css中为`.main-content`添加`display: flex`和`flex-direction: column`两条规则，为`.thumbnail-list`添加`flex`属性，并为`.detail-image-container`添加新的样式声明。

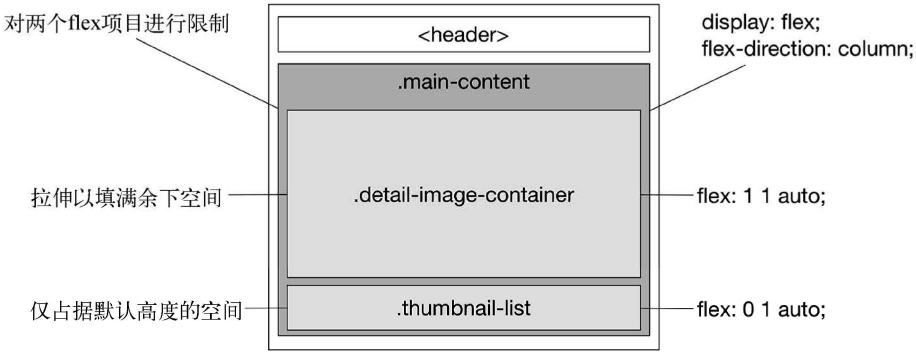


图4-15　将`.main-content`转换为flex容器

```
...
.main-content {
  flex: 1 1 auto;
  display: flex;
  flex-direction: column;
}

...

.thumbnail-list {
  flex: 0 1 auto;
  list-style: none;
  padding: 0;

  white-space: nowrap;
  overflow-x: auto;
}

...

.thumbnail-title {
  ...
}

.detail-image-container {
  flex: 1 1 auto;
}

.detail-image {
  ...
```

有人可能会好奇，为什么不像对`.detail-image`一样为`.thumbnail-list`和`.detail-image-container`设置百分比高度，比如将`.thumbnail-list`的高度设置为25%，将`.detail-image-wrapper`的高度设置为75%。从表面上看，这样做是符合逻辑的，但最终结果并不会如你所愿。在`.detail-image`宽度的作用下，`.detail-image-container`可能偏大，而`.thumbnail-list`在不同尺寸的窗口中也可能偏大或偏小。

简而言之，使用flex属性设置flex项目尺寸，并设置比较重要的某一固定尺寸（如.detail-image的宽度）才是正确的方式。

接下来看看如何将缩略图列表放到大图下面。默认情况下，flex项目按照它们在HTML中的顺序依次绘制。这被称为**源顺序**（source order），是开发者控制元素绘制顺序的主要手段。

一种改变源顺序的方法是剪切大图的那段代码并将其粘贴到.thumbnail-list前面，但其实通过一个新的flexbox属性也能解决问题。

在styles.css中，为.thumbnail-list添加如下一条order样式声明：

```
...
.thumbnail-list {
  flex: 0 1 auto;
  order: 2;
  list-style: none;
  padding: 0;

  white-space: nowrap;
  overflow-x: auto;
}
...
```

order属性值可以是任意整数。默认值为0，意味着使用源顺序；包括负数在内的其他值则告诉浏览器将该flex项目置于其他项目之前或之后。为.thumbnail-list设置的order: 2告诉浏览器将它绘制在其他order值更小的元素**后面**——如.detail-image-container，其order值为默认值0。

保存styles.css，查看浏览器。可以看到，缩略图现在已经在页面底部了（如图4-16所示）。

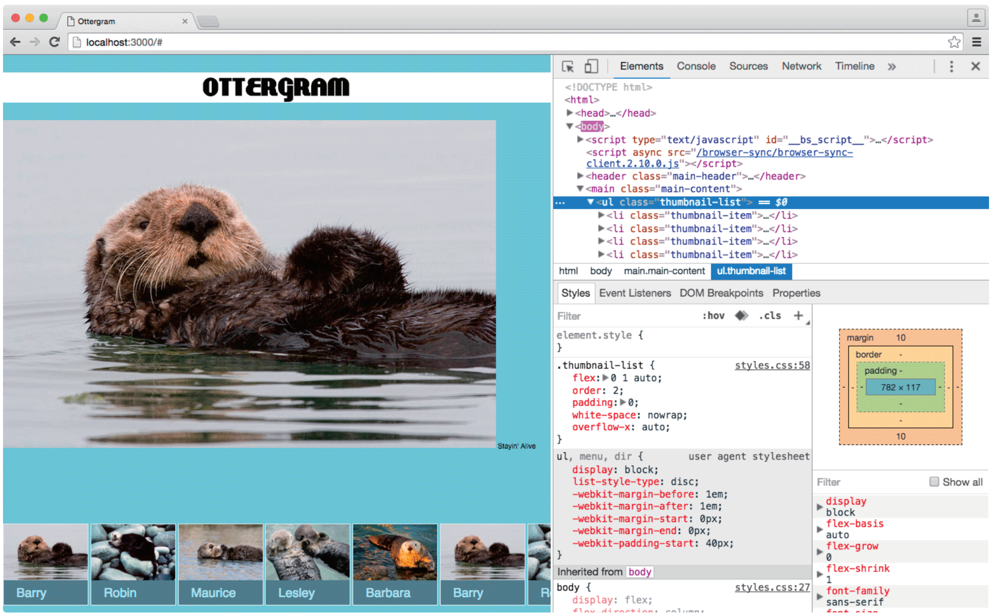


图4-16 改变元素的绘制顺序

在后面继续对Ottergram进行UI布局的过程中还会用到`display: flex`。到目前为止，我们只尝试过仅有数个元素的flex容器。试着将`.thumbnail-list`变成flex容器，体验更多关于flexbox的内容。

```
...
.thumbnail-list {
  flex: 0 1 auto;
  order: 2;
  display: flex;
  list-style: none;
  padding: 0;

  white-space: nowrap;
  overflow-x: auto;
}
...
```

保存文件，如果看到如图4-17所示的奇怪效果，先不要慌。

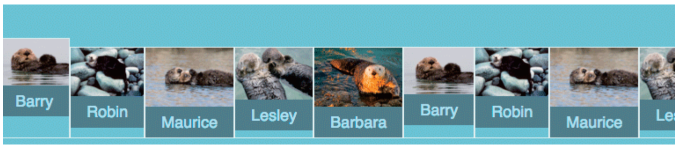


图4-17　排列不齐的缩略图

为修复问题，将`.thumbnail-item`的`width`声明替换为`min-width`和`max-width`两个声明，这样便可以修复因图片尺寸不一导致的布局问题。

还可以删除为`.thumbnail-item + .thumbnail-item`元素设置的`margin-top`声明，现在的布局完全不需要它。

```
...
.thumbnail-item + .thumbnail-item {
  margin-top: 10px;
}

.thumbnail-item {
  display: inline-block;
  width: 120px;
  min-width: 120px;
  max-width: 120px;
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);
}
...
```

接下来看看如何处理`.thumbnail-list`内部的flex项目之间的间隙问题。在styles.css中，为`.thumbnail-list`类添加一个`justify-content`声明。

```
...
.thumbnail-list {
```

```
  flex: 0 1 auto;
  order: 2;
  display: flex;
  justify-content: space-between;
  list-style: none;
  padding: 0;

  white-space: nowrap;
  overflow-x: auto;
}
...
```

`justify-content`属性决定flex容器如何控制flex项目在主轴方向上的绘制方式。使用`space-between`值，保证每个flex项目之间的空隙是相等的。

`justify-content`属性可以取5种不同的值，图4-18展示了不同取值的效果。

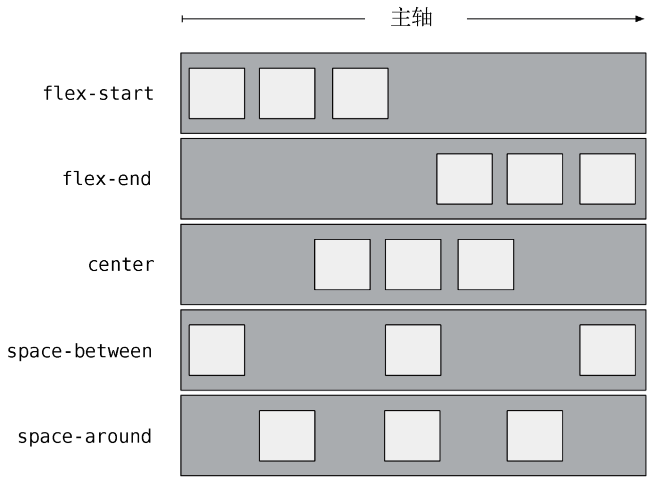


图4-18 `justify-content`效果示意图

`.thumbnail-list`布局已经完成，下面来处理`.detail-image-container`及其内容。

4.2.6 居中显示大图

大图是Ottergram页面的焦点，应当处于最突出的位置，方便用户好好欣赏可爱的水獭。另外，它还需要一个漂亮的标题。

要让大图居中，首先需要将图片和标题放入一个容器，进而将`.detail-image-container`中的**包装元素**居中，如图4-19所示。

在`.detail-image-container`中让`.detail-image`居中虽然简单，但想要给`.detail-image-title`设置正确的偏移量就比较麻烦，因为`.detail-image`和`.detail-image-container`会动态改变。

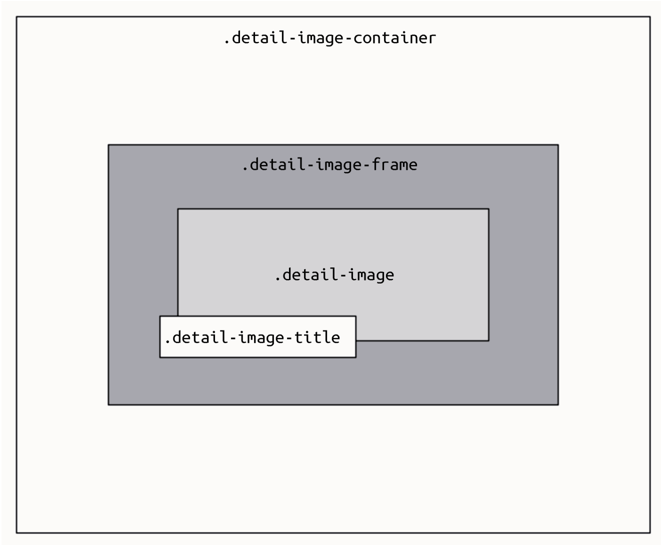


图4-19　将.detail-image和.detail-image-title置入“相框”

这时，一个中间的包装元素就很有用了——它会限制.detail-image的尺寸，并作为.detail-image-title的定位参照。

在index.html中添加一个类名为detail-image-frame的<div>标签。

```
...
    </ul>

      <div class="detail-image-container">
        <div class="detail-image-frame">
          <img class="detail-image" src="img/otter1.jpg" alt="">
          <span class="detail-image-title">Stayin' Alive</span>
        </div>
      </div>
    </main>
  </body>
</html>
```

保存index.html，然后在styles.css中为.detail-image-frame类添加一句简单的样式声明text-align: center。这是一种无须flexbox即可实现居中的方式——但注意，只能是水平居中。

```
...
.detail-image-container {
  flex: 1 1 auto;
}

.detail-image-frame {
  text-align: center;
}
```

```
.detail-image {
  width: 90%;
}
```

紧接着要让.detail-image-frame在.detail-image-container内部居中。更新样式文件，将.detail-image-container设置为flex容器，使用justify-content: center使其flex项目沿主轴方向（在本例中，也就是默认的水平方向）居中，添加一个新的flexbox属性声明align-items: center，使flex项目在侧轴方向（垂直方向）上居中。

```
...
.detail-image-container {
  flex: 1 1 auto;
  display: flex;
  justify-content: center;
  align-items: center;
}
...
```


保存文件，再看下效果，水獭已经稳稳当当地躺在.detail-image-container正中央了（如图4-20所示）。

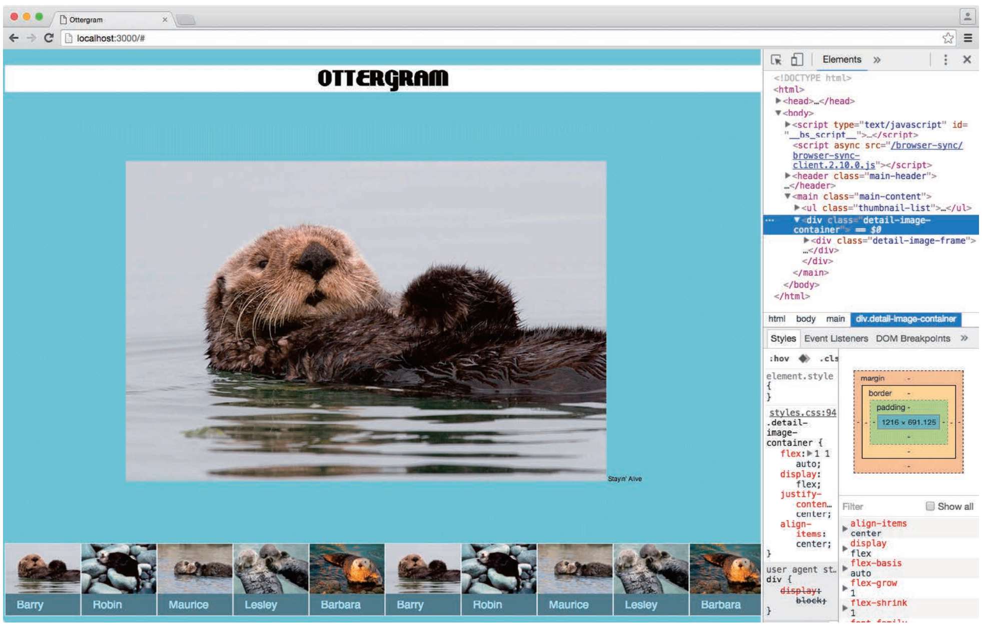


图4-20　.detail-image-frame在.detail-image-container里居中显示

4.3　绝对定位与相对定位

有时候可能需要将元素放在另一元素中的精确位置上，这可以使用CSS提供的**绝对定位**实现。

使用绝对定位将.detail-image-title放在.detail-image-frame的左下角，如图4-21所示。

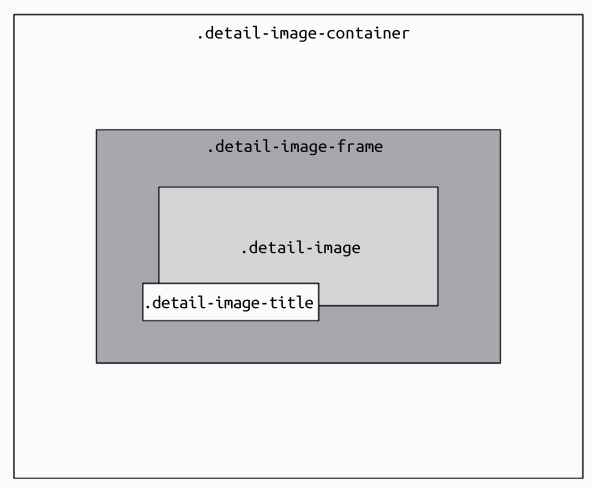


图4-21　绝对定位.detail-image-title

绝对定位元素必须满足以下3个条件。

- position属性值为absolute，告知浏览器该元素将脱离**常规文档流**（normal flow），不与其兄弟元素一起布局。
- 使用top、right、bottom或left中的一个或多个属性设置坐标，坐标值可以是绝对长度（如像素值）或相对长度（如百分比值）。
- 有一个position属性显示声明为relative或absolute的祖先元素。这点很重要，若不满足本条件，该绝对定位元素将会相对于<html>元素（浏览器窗口）定位。

一条忠告：虽然所有元素都用绝对定位看似还挺不错的，但必须少这么用。因为一个全部由绝对定位组成的页面基本无法维护，而且在其他屏幕上的效果可能会很糟糕。

指定定位坐标实际上是在指定元素边缘与容器边缘之间的距离，如图4-22所示。

图4-22展示了两个绝对定位的例子：第一个例子中的元素定位为上边缘距容器顶部50px，左边缘距容器左边缘200px；第二个例子有所不同，根据下边缘和左边缘定位。

在定位.detail-image-title之前，需要先将.detail-image-frame的position属性设置为relative，然后相对.detail-image-frame定位.detail-image-title。

```
...
.detail-image-frame {
  position: relative;
  text-align: center;
}
...
```

为.detail-image-frame设置position: relative是因为既需要将它保留在常规文档流

中，又需要它作为绝对定位元素的容器，因此其`position`属性必须显式定义。

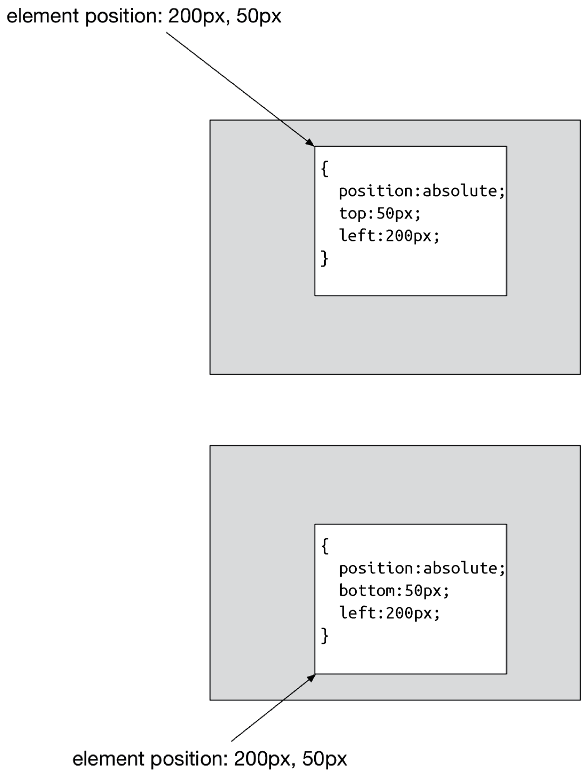


图4-22 元素基于边缘绝对定位

在styles.css的底部为`.detail-image-title`添加声明，目前将颜色设为白色并将字体设置为默认大小的四倍即可。

```
...
.detail-image {
  width: 90%;
}

.detail-image-title {
  color: white;
  font-size: 40px;
}
```

目前一切都很好（如图4-23所示），但还是太普通了。

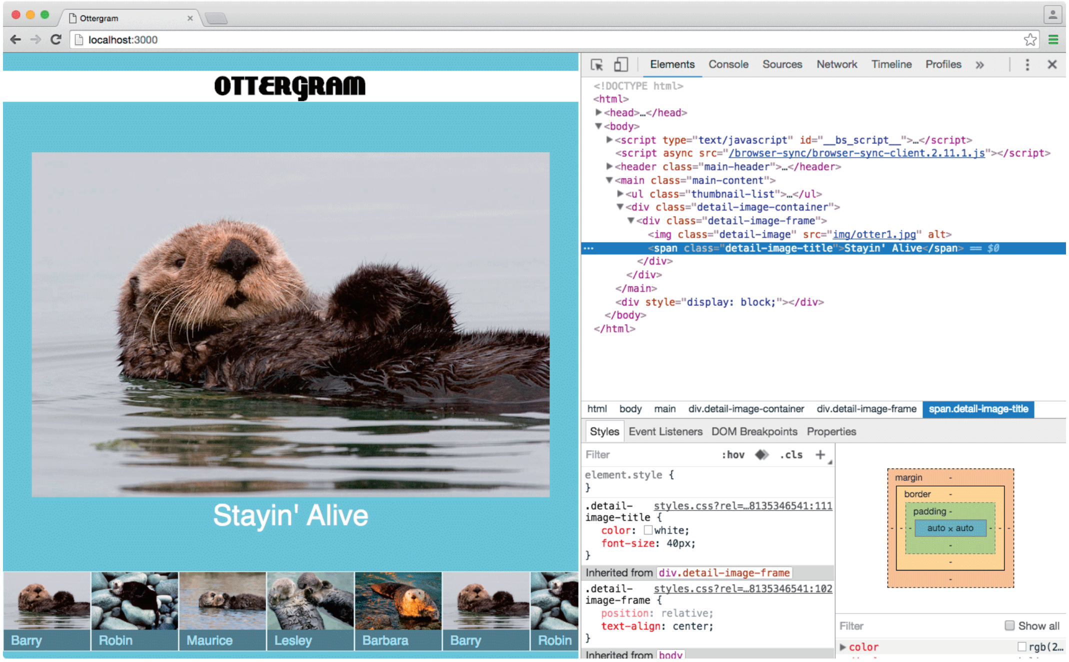


图4-23　`.detail-image-title`的基本样式

为深入接触CSS，我们来试着为`.detail-image-title`添加一些文本特效。记住，定位带有样式的文本时，元素盒子可能会因为自定义字体或其他特效的视觉因素而改变。本例会在定位`.detail-image-title`之前先设置其文字样式。在styles.css中为`.detail-image-title`添加`text-shadow`属性。

```
...
.detail-image-title {
  color: white;
  text-shadow: rgba(0, 0, 0, 0.9) 1px 2px 9px;
  font-size: 40px;
}
```

顾名思义，`text-shadow`属性就是为文本添加阴影。它接受一个颜色值作为阴影颜色，一对**偏移量**（决定阴影的上下左右位置），以及一个**模糊半径值**（该值是可选的，值越大则阴影越大，颜色越亮）。

我们将阴影颜色值设为`rgba(0, 0, 0, 0.9)`，这将呈现略微有些透明的黑色。阴影向右偏移1px，向下偏移2px（负值会使阴影向左或上偏移）。最后的9px是模糊半径。图4-24展示了阴影效果。

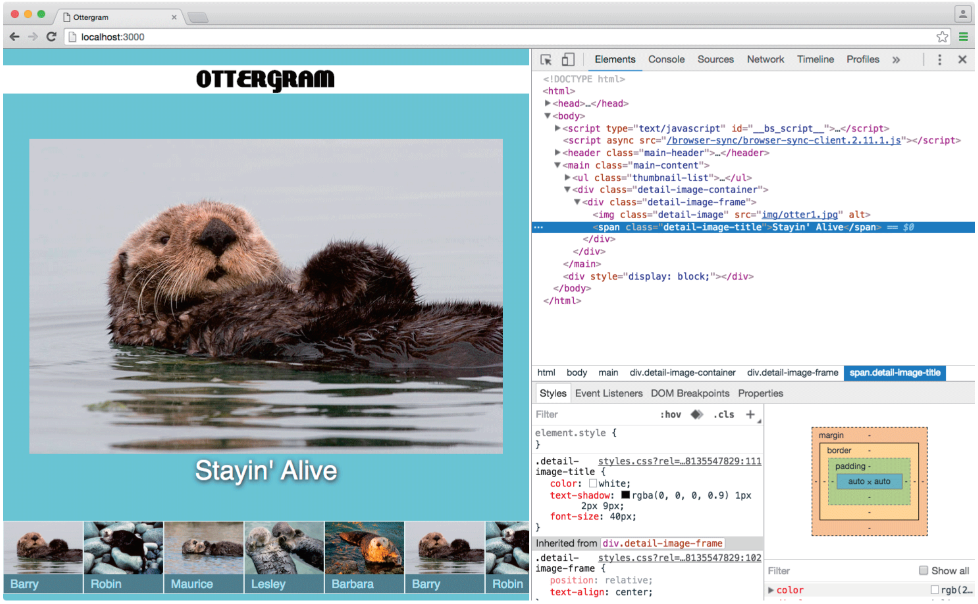


图4-24　.detail-image-title的文字阴影效果

可以在开发者工具的Elements面板中调整`text-shadow`值，感受一下`text-shadow`的工作方式（如图4-25所示）。

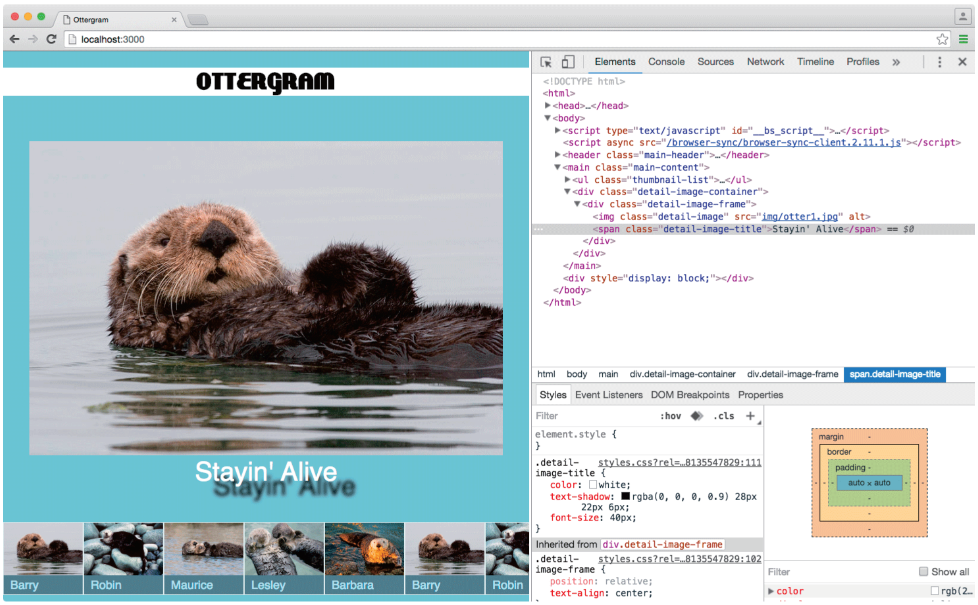


图4-25　夸张的文字阴影效果

最后再锦上添花，添加一种自定义字体。正如第3章所做的那样，在styles.css中添加@font-face声明，并将Airstream字体添加至项目中。然后为.detail-image-title设置font-family: airstreamregular。

```
@font-face {
    font-family: 'airstreamregular';
    src: url('fonts/Airstream-webfont.eot');
    src: url('fonts/Airstream-webfont.eot?#iefix') format('embedded-opentype'),
         url('fonts/Airstream-webfont.woff') format('woff'),
         url('fonts/Airstream-webfont.ttf') format('truetype'),
         url('fonts/Airstream-webfont.svg#airstreamregular') format('svg');
    font-weight: normal;
    font-style: normal;
}

@font-face {
    font-family: 'lakeshore';
    ...
}

...

.detail-image-title {
  font-family: airstreamregular;
  color: white;
  text-shadow: rgba(0, 0, 0, 0.9) 1px 2px 9px;
  font-size: 40px;
}
```

到目前为止，效果已经非常酷了（如图4-26所示）。

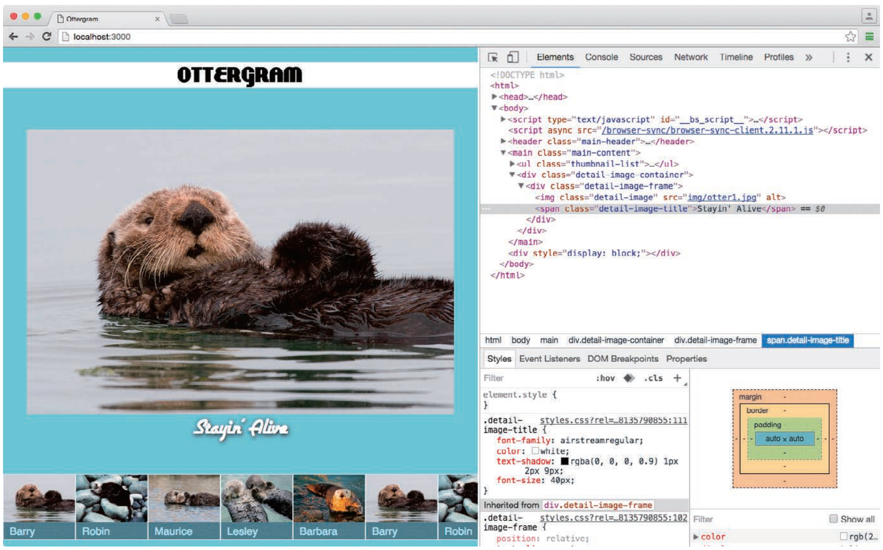


图4-26　更漂亮的效果

既然.detail-image-title的样式设置已经完成，就可以为其设置position: absolute，使浏览器将其置于.detail-image-frame中的精确位置上。指定bottom: －16px以及left: 4px，将.detail-image-title放在.detail-image-frame的底边之下、左边之内的位置。（坐标值可以为负数。）

```
...
.detail-image-title {
  position: absolute;
  bottom: -16px;
  left: 4px;

  font-family: airstreamregular;
  color: white;
  text-shadow: rgba(0, 0, 0, 0.9) 1px 2px 9px;
  font-size: 40px;
}
```

保存样式文件，可以在浏览器中看到.detail-image-title现在已在图片底部偏左的位置。Ottergram的样式已十分别致了（如图4-27所示）。

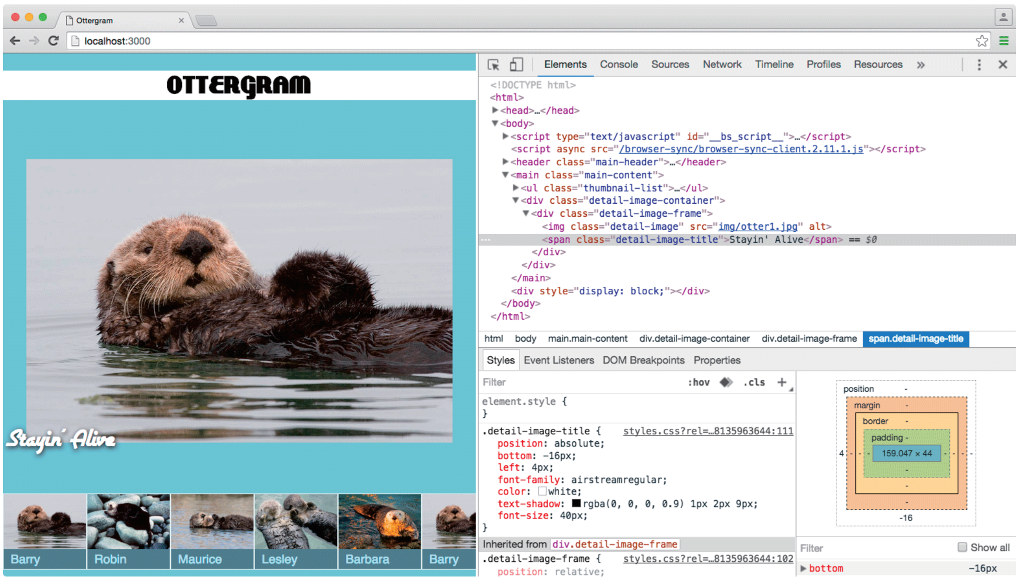


图4-27　效果极好

稍事休息，好好欣赏下劳动成果吧。添加flexbox样式之后的Ottergram已是动态流体布局了。下一章将学习如何让布局根据窗口尺寸自适应。

第 5 章

使用媒体查询完成自适应布局

本章将介绍一种能够根据浏览器窗口大小等特征切换样式的技术。仅需使用少量代码，就可在大屏上展示一种不同的布局。而随着浏览器窗口的大小变化，页面布局无须刷新，也能实时切换。图5-1展示了两种不同情况下的布局。

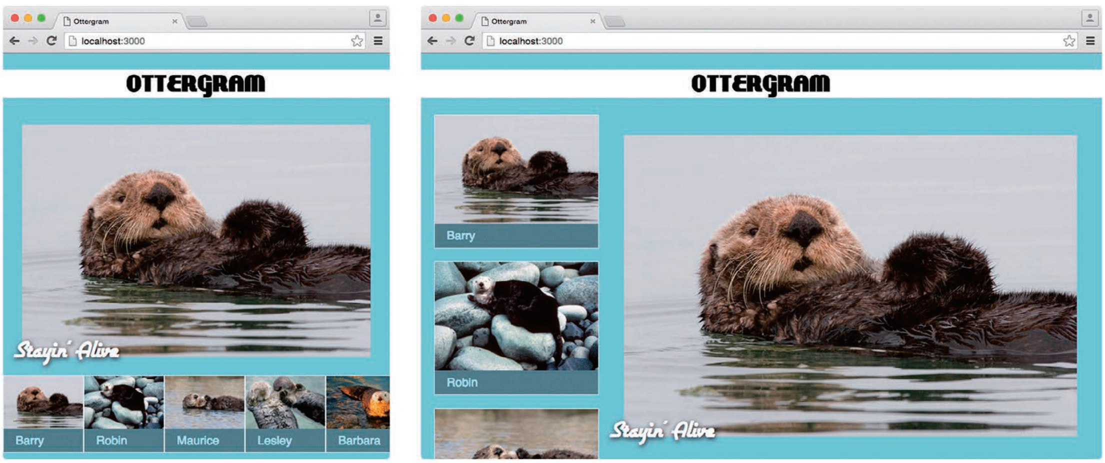


图5-1　两种布局

这种行为的行业术语叫**响应式站点**（responsive website）。不幸的是，该术语常常会造成误解——一些人可能认为它的意思是“速度很快的网站”或者“有视觉动画的网站”。因此还是使用**自适应布局**（adaptive layout）这个术语比较好。

根据当前浏览器的条件添加备用样式的方式有很多种。推荐方式是先为最小屏幕书写样式，随后使用**媒体查询**，当**视口**（浏览器可见区域）大小超出一定阈值时，使用添加的重载样式。

对传统浏览器（如开发Ottergram时使用的浏览器）来说，视口即浏览器窗口所呈现的部分，很直观。但在移动浏览器上就复杂一些，因为移动浏览器有多种视口，每一种在页面渲染过程中都起作用。

前端开发者需要关注的是**布局视口**（layout viewport，有时也称作**真实视口**）。布局视口告诉浏览器："假设我的实际宽度为980像素，然后再绘制页面。"

而用户更关心的则是**视觉视口**（visual viewport），他们在这上面放大、缩小页面（如图5-2所示）。



图5-2 视觉视口和布局视口

如果现在就拿手机查看Ottergram，看到的效果就会像图5-2那样，首先出现页面左上角的放大部分。毋庸多言，就算用户能够手动缩小，也不会有人愿意看到这种默认效果。

此前提到过开发Ottergram的思想是移动优先，大体确是如此。标记和样式都是按照移动友好的原则——使用最少的标记，先为最底层元素添加样式——进行编写的。现在需要告诉浏览器应当使用怎样的布局视口。

5.1 重置视口

在第3章中，我们为Ottergram添加了normalize.css，确保不同浏览器的默认样式是相同的。有了这些默认样式，才能更自信地编写自己的CSS，因为我们知道它在不同浏览器中的展现是一致的。

针对布局视口，也需要做一些类似的事。正如不同浏览器的默认样式可能不尽相同，每个

浏览器都可能有自己的默认布局视口。不过，不像使用normalize.css那样，我们不会将所有浏览器的视口都设置成同一个值，而是使用一个`<meta>`标签告诉浏览器使用最佳尺寸展现Ottergram。

在index.html中加入一个`<meta>`标签，告知浏览器布局视口的宽度与设备屏宽相同。同时将`initial-scale`值设置为`1`，使页面缩放值为100%。

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet"
      href="https://cdnjs.cloudflare.com/ajax/libs/normalize/3.0.3/normalize.min.css">
    <link rel="stylesheet" href="stylesheets/styles.css">
    <title>Ottergram</title>
  </head>
...
```

记得保存文件。我们将布局视口设置成了**理想视口**，即浏览器厂商推荐的、某种设备上的最佳视口尺寸。理想视口有很多种尺寸，因为设备种类和浏览器类型太多了。

表5-1对不同类型的视口进行了总结。

表5-1　不同视口概览

视　　口	描　　述	设备类型
视口	等同于浏览器窗口区域，作为`<html>`元素的容器	台式机、笔记本电脑
布局视口	虚拟屏幕，大于设备实际屏幕，用于计算页面布局	移动设备
视觉视口	用户在设备屏幕上看到的可缩放区域，缩放对页面布局不会产生影响	移动设备
理想视口	特定设备上的特定浏览器的最佳尺寸	移动设备

启动browser-sync，打开 Chrome的开发者工具。找到Elements菜单项左侧的Toggle Device Mode（切换设备模式）按钮。图5-3展示了按钮的位置。

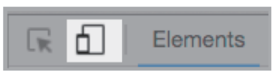


图5-3　Toggle Device Mode按钮

点击按钮，激活**设备模式**。可以看到，Ottergram现在显示在一个模拟的智能手机上，还有一个用于选择不同手机类型和屏幕尺寸的菜单。点击预设尺寸下面的灰色区域，就可以在大中小尺寸之间切换。此外，还有一个快速选择屏幕方向的按钮。图5-4展示了本书创作时在设备模式下的截图，你看到的效果可能有所不同，因为开发者工具会经常更新。



图5-4 使用设备模式进行响应式测试

有了<meta>元素后，Ottergram便能够在智能手机这样的小屏幕上完整展示。而在屏幕更大一些的设备上（如平板电脑、笔记本电脑等）使用另一种不同的布局可能更合适。接下来将结合flexbox和媒体查询技术，应用不同的布局样式。

在继续之前，请再次点击按钮，退出设备模式。

5.2 添加媒体查询

媒体查询允许我们将CSS声明进行分组，指定应用这些样式的条件。这些条件可能类似于“屏幕最低宽度为640px”或“屏幕宽度大于高度且拥有高像素密度”。

媒体查询语法以`@media`开头，接下来是匹配条件，再接着是一组大括号，括号里面是整个样式声明。来看下它究竟是什么样的。

先从styles.css末尾添加媒体查询。当设备宽度在768px（通常是平板电脑的宽度）及以上时，我们新增的媒体查询样式就会生效。

```
...
.detail-image-title {
  ...
}

@media all and (min-width: 768px) {
  /* 此处放置样式*/
}
```

`@media`之后是**媒体类型**`all`。媒体类型一开始是用于区分不同设备的，如智能电视、手持设备等。不幸的是，浏览器并没有精确实现，所以只能指定为`all`。唯一不用指定`all`的情况是需要设置打印样式时，这时候可以放心使用`print`媒体类型。

在媒体类型之后是使用样式的条件，这里使用的是`min-width`。这条件看起来很像样式声明。

为实现图5-1所示的效果，需要改变`.main-content`元素的`flex-direction`，这样就会让缩略图列表与大图并排显示。我们不希望缩略图列表导致浏览器滚动，而是独立于浏览器窗口滚动，因此需要添加`overflow: hidden`声明。

在styles.css末尾的媒体查询中添加如下样式：

```
...
@media all and (min-width: 768px) {
  /*此处放置样式*/
  .main-content {
    flex-direction: row;
    overflow: hidden;
  }
}
```

当你保存文件并拉宽浏览器窗口触发媒体查询时，可能会被结果吓到，因为目前页面看起来应该像图5-5那样。不必担心，几行代码就可以解决问题。

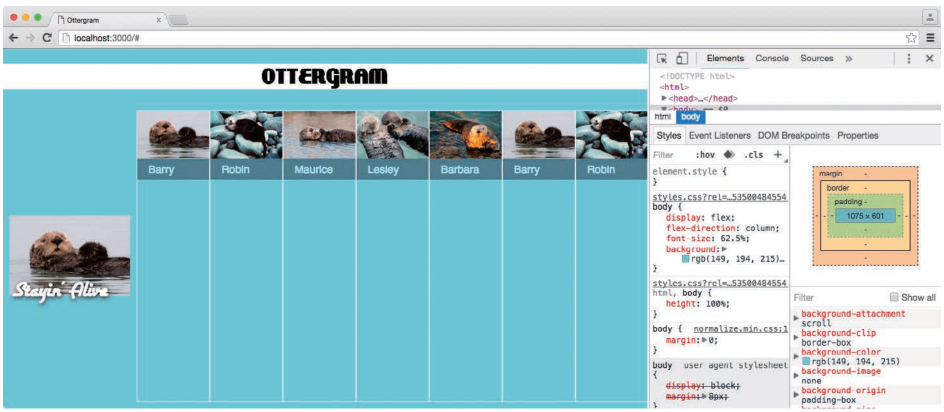


图5-5　混乱的页面

缩略图需要排成一列，而不是一行。这操作起来很简单，因为你使用的是flexbox布局。在媒体查询中加入一个CSS声明语句块，将`.thumbnail-list`的`flex-direction`设置为`column`。

```
...
@media all and (min-width: 768px) {
  .main-content {
    flex-direction: row;
    overflow: hidden;
  }

  .thumbnail-list {
    flex-direction: column;
  }
}
```

保存styles.css，情况大有改善（如图5-6所示）!

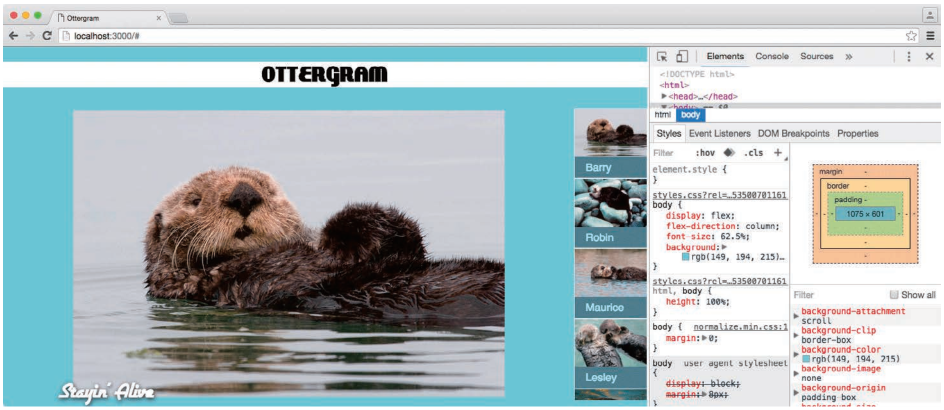


图5-6 将`flex-direction`设置为`column`之后

根据设计，缩略图列表应该在左侧，修改`.thumbnail-list`的`order`样式可以实现该效果。此前我们将`order`设置为2，使`.thumbnail-list`显示在`.detail-image-container`下方。现在，在styles.css的媒体查询中将`order`设置为0，以便它遵循源顺序，这样一来位置就对了。

```
...
@media all and (min-width: 768px) {
  .main-content {
    flex-direction: row;
    overflow: hidden;
  }

  .thumbnail-list {
    flex-direction: column;
    order: 0;
  }
}
```

保存文件并确认缩略图列表出现在页面左侧。

差不多就要完成啦！在styles.css中给.thumbnail-list和.thumbnail-item添加一些样式，使尺寸和空隙显得更漂亮一些。

```
...
@media all and (min-width: 768px) {
  .main-content {
    flex-direction: row;
    overflow: hidden;
  }

  .thumbnail-list {
    flex-direction: column;
    order: 0;
    margin-left: 20px;
  }

  .thumbnail-item {
    max-width: 260px;
  }

  .thumbnail-item + .thumbnail-item {
    margin-top: 20px;
  }
}
```

再次保存文件，切换到浏览器。现在无论浏览器窗口是宽是窄，整个布局都一样炫酷（如图5-7所示）。

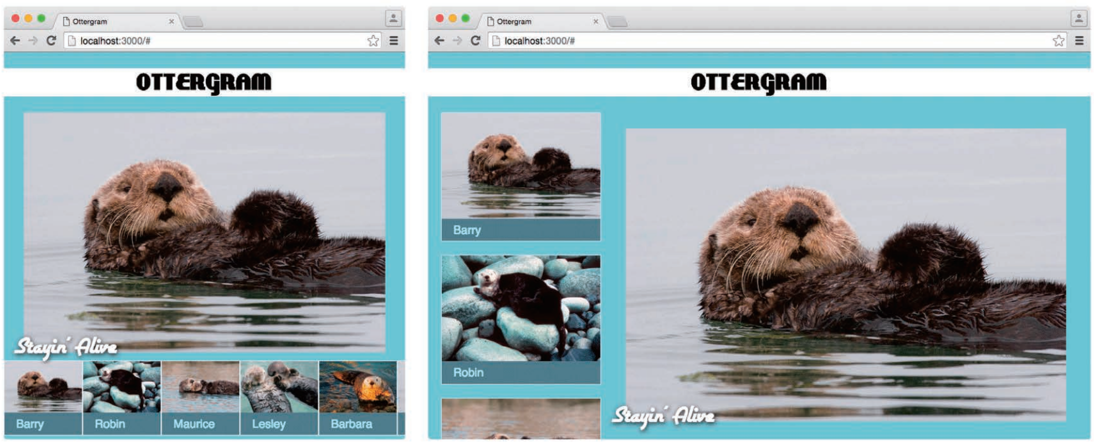


图5-7　响应式布局

Ottergram项目正在稳步前进！我们已经创建了一个又美观、又可以适应多种屏幕尺寸的网页。下一章开始使用JavaScript为项目添加交互功能。

5.3　初级挑战：屏幕方向

当前的媒体查询是根据视口宽度来改变布局的。我们可以将视口分为两类：高度大于宽度的视口和宽度大于高度的视口。这就是视口可能的两种**方向**（orientation）模式。

到MDN中查找媒体查询的相关文档，并更新媒体查询，使布局根据屏幕方向而非宽度改变。

5.4　延展阅读：flexbox 布局通用解决方案与 bug

Philip Walton是一位开发者，他维护了两份非常重要的flexbox资源。第一份是Solved by Flexbox网站（philipwalton.github.io/solved-by-flexbox），该网站提供了用flexbox实现常见布局的demo以及创建它们所需的信息。不用flexbox的话，其中某些布局真的很难实现。

第二份资源是Flexbugs，地址是github.com/philipwalton/flexbugs。flexbox很棒，但也并非完美。Flexbugs提供了开发者使用flexbox时遇到的常见问题的解决方案和变通办法。这些信息由社区中遇到问题的开发者提供，列表也得到了很好的维护。

5.5　高级挑战：圣杯布局

在开始本挑战前，请先确保已复制了一份代码！完成本挑战需要对标记和样式进行重大改动，所以请使用副本代码完成本挑战，原版代码还得留着继续下一章的学习。

参考Solved by Flexbox，在Ottergram中实现圣杯布局（Holy Grail layout）。创建带缩略图的第二个导航栏，并将其放在视口的另一侧。

记得在页面底部添加页脚。使用`<footer>`标签，并在其内部加入一个`<h1>`。

第 6 章

JavaScript事件处理

你知道水獭有一项很酷的技能吗？睡觉的时候，为了防止漂走，它们会手牵着手。在学习JavaScript事件回调的时候，在脑海中想象这个场景就好了。

JavaScript是一门通过操作页面中的DOM元素和CSS样式来为网站添加交互的编程语言。起初，它是为非专业编程人员设计的。然而随着发展壮大，如今它已被应用于多种应用开发，如访问Gmail或者Netflix等站点的时候，你就是在与JavaScript程序进行交互。事实上，Atom文本编辑器也是一个用JavaScript编写的桌面应用。

尽管JavaScript功能强大并且使用广泛，不过像其他编程语言一样，它也有不足。但随着不断开发Ottergram和本书中的其他项目，你会逐渐掌握如何对JavaScript用其所长，避其所短。

JavaScript有不同的版本，本书中的项目涉及其中三个版本，均为ECMAScript（ES）标准规范的修订版。表6-1对这几个版本进行了总结。

表6-1　本书中使用的JavaScript版本

ECMAScript版本	发布日期	备　注
3	1999年12月	得到最广泛支持的版本；囊括了变量、类型、函数等大部分你会用到的语言特性
5	2009年12月	向后兼容，新增了一些语法特性，如可预防语言中易出错用法的**严格模式**
6	2015年6月	包含新的语法和语言特性；在本书写作之时，多数浏览器尚未支持ES6，不过 ES6代码可以转换为ES5代码，从而能够在多数浏览器上使用

本章将使用JavaScript来让Ottergram拥有交互性：当用户点击一个缩略图的时候，大图和大图标题都会发生变化。

为了达成这个效果，首先需要编写一个读取图片URL并将其展示在大图区域的JavaScript**函数**（其中包含了一系列浏览器需要执行的步骤）。接着，要确保缩略图被点击时这个函数会运行。当然，也可以另外写一个函数来隐藏大图区域，并且在按下Esc键时运行这个函数。

本章结束后，Ottergram将能够展示每一张水獭的大图（如图6-1所示）。

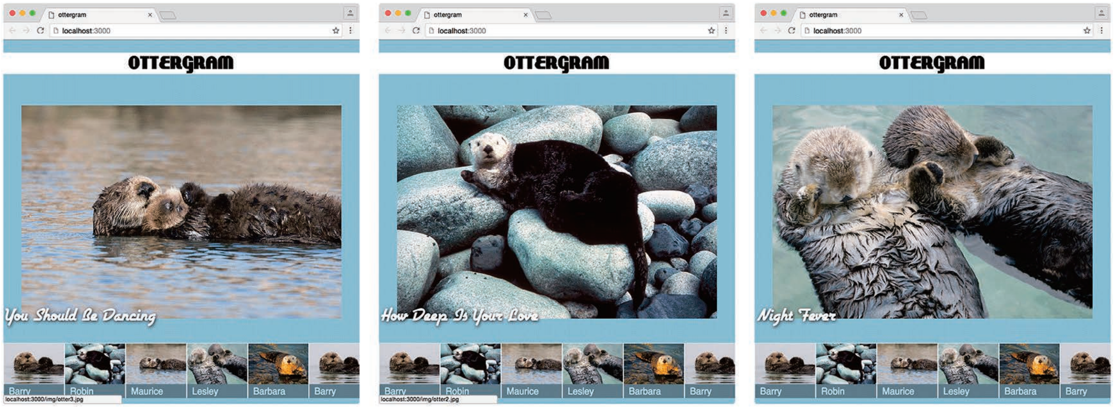


图6-1　点击缩略图改变大图和大图标题

编写这些函数时，将会用到一系列与页面交互的浏览器预定义接口。这样的预定义接口有很多，这里只会涉及与当前任务相关的几个。如果感兴趣，可以在MDN（developer.mozilla.org/en-US/docs/Web/API/Element）中了解更全面的内容。

6.1　准备锚标签

在添加JavaScript交互功能之前，需要对标记进行少量的修改。虽然现在缩略图被锚标签包裹着，但实际上这些标签没有链接到任何资源，而是使用#作为href属性的值，让浏览器停留在同一个页面。为了使点击缩略图能够引发一些有趣的变化，需要在这里做一些改动。

首先，除保留index.html中的5个.thumbnail-item元素外，删掉其余全部内容——现在不再需要这些重复的内容了，因为当前布局已经不错了。

然后，修改锚标签的href属性的值，不再使用#，而是设置为每个<img>标签对应的src的值。

这种对HTML的重复修改可以借助Atom来完成。与其他的文本编辑器一样，Atom也有查找并替换文本的功能。选择Find → Find in Buffer或者使用快捷键Commond + F（Control + F），在 编辑器窗口的底部打开Find in Buffer面板（如图6-2所示）。



图6-2　使用Atom的查找替换功能

在Find in current buffer文本框中输入#，在Replace in current buffer文本框中输入img/otter.jpg，然后点击右下角的Replace All按钮。

这一步会将所有<a href="#">标签修改为<a href="img/otter.jpg">。现在只需手动给每个标签加上相应的序号就可以了（如img/otter1.jpg、img/otter2.jpg等）。

按下Esc键关闭Find in Buffer面板。index.html应如下所示：

```
...
      <ul class="thumbnail-list">
        <li class="thumbnail-item">
          <a href="#">
          <a href="img/otter1.jpg">
            <img class="thumbnail-image" src="img/otter1.jpg" alt="">
            <span class="thumbnail-title">Barry</span>
          </a>
        </li>
        <li class="thumbnail-item">
          <a href="#">
          <a href="img/otter2.jpg">
            <img class="thumbnail-image" src="img/otter2.jpg" alt="">
            <span class="thumbnail-title">Robin</span>
          </a>
        </li>
        <li class="thumbnail-item">
          <a href="#">
          <a href="img/otter3.jpg">
            <img class="thumbnail-image" src="img/otter3.jpg" alt="">
            <span class="thumbnail-title">Maurice</span>
          </a>
        </li>
        <li class="thumbnail-item">
          <a href="#">
          <a href="img/otter4.jpg">
            <img class="thumbnail-image" src="img/otter4.jpg" alt="">
            <span class="thumbnail-title">Lesley</span>
          </a>
        </li>
        <li class="thumbnail-item">
          <a href="#">
          <a href="img/otter5.jpg">
            <img class="thumbnail-image" src="img/otter5.jpg" alt="">
            <span class="thumbnail-title">Barbara</span>
          </a>
        </li>
      </ul>
...
```

要让JavaScript可以使用锚元素，还得为它们添加额外的属性。当使用CSS控制样式时，通过类名选择器来关联页面中的元素，而JavaScript则使用数据（`data`）**属性**。

与你使用过的其他HTML属性一样，数据属性对浏览器而言也没有什么特殊的含义，这与之前使用过的`src`和`href`不同。数据属性唯一的要求就是属性名要以`data-`开始。使用自定义的数据属性便于指定JavaScript与哪个HTML元素进行交互。

从技术上讲，在JavaScript中可以使用类名来访问页面元素，数据属性也可以用于样式选择器——但不推荐这样做，因为JavaScript和CSS不依赖相同属性的话，页面会更好维护。

请按下面给出的代码来更新index.html中锚标签的数据属性。请注意，代码中的换行是为了

在任何页面中都可以正确地展示。你可以根据喜好来选择是否换行，这不会对浏览器造成影响。

```
...
        <li class="thumbnail-item">
          <a href="img/otter1.jpg" data-image-role="trigger"
                                   data-image-title="Stayin' Alive"
                                   data-image-url="img/otter1.jpg">
            <img class="thumbnail-image" src="img/otter1.jpg" alt="">
            <span class="thumbnail-title">Barry</span>
          </a>
        </li>
          <li class="thumbnail-item">
            <a href="img/otter2.jpg" data-image-role="trigger"
                                     data-image-title="How Deep Is Your Love"
                                     data-image-url="img/otter2.jpg">
            <img class="thumbnail-image" src="img/otter2.jpg" alt="">
            <span class="thumbnail-title">Robin</span>
          </a>
        </li>
        <li class="thumbnail-item">
          <a href="img/otter3.jpg" data-image-role="trigger"
                                   data-image-title="You Should Be Dancing"
                                   data-image-url="img/otter3.jpg">
            <img class="thumbnail-image" src="img/otter3.jpg" alt="">
            <span class="thumbnail-title">Maurice</span>
          </a>
        </li>
        <li class="thumbnail-item">
          <a href="img/otter4.jpg" data-image-role="trigger"
                                   data-image-title="Night Fever"
                                   data-image-url="img/otter4.jpg">
            <img class="thumbnail-image" src="img/otter4.jpg" alt="">
            <span class="thumbnail-title">Lesley</span>
          </a>
        </li>
        <li class="thumbnail-item">
          <a href="img/otter5.jpg" data-image-role="trigger"
                                   data-image-title="To Love Somebody"
                                   data-image-url="img/otter5.jpg">
            <img class="thumbnail-image" src="img/otter5.jpg" alt="">
            <span class="thumbnail-title">Barbara</span>
          </a>
        </li>
...
```

为大图添加数据属性，如下：

```
...
        <div class="detail-image-container">
          <div class="detail-image-wrapper">
      <img class="detail-image" data-image-role="target" src="img/otter1.jpg" alt="">
      <span class="detail-image-title" data-image-role="title">Stayin' Alive</span>
          </div>
        </div>
...
```

6

JavaScript代码可以通过这些数据属性来访问页面中的特定元素，因为浏览器允许JavaScript查询网页的内容。举例来说，你可以找到匹配某个选择器的所有内容，选择器的形式如`data-image-role="trigger"`。一旦查询找到了匹配项，就会返回匹配元素的引用。

取得一个元素的引用之后，就可以对这个元素进行很多操作：读取或修改它的属性值、改变它内部的文本，甚至访问它周围的元素。当通过引用来修改元素的时候，浏览器会立即更新页面。

本章将使用JavaScript代码来获取锚元素和大图元素的引用，读取锚的数据属性，并且改变大图的`src`属性的值——这就是Ottergram交互功能的原理。

你可能注意到了，这些锚标签以及大图的`<img>`标签都包含`data-image-role`属性，不过它们的值是不同的。

虽然不是必须为锚标签和`<img>`标签使用相同的数据属性名，但是推荐这么做，因为可以提醒开发者这些元素具有相同的JavaScript行为。

在HTML中最后一项必需的改动是：告诉HTML执行JavaScript。通过在index.html中添加`<script>`标签来实现这一点，这个`<script>`标签指向即将创建的scripts/main.js文件。

```
...
          </div>
        </div>
      </main>
    <script src="scripts/main.js" charset="utf-8"></script>
  </body>
</html>
```

当浏览器发现`<script>`标签时，它就会立即运行引用文件中的代码。JavaScript不能在浏览器渲染HTML之前访问HTML中的元素，因此将`<script>`放到`body`的底部可以保证JavaScript在所有的标记被解析之后再运行。

现在HTML已经做好了与即将编写的JavaScript关联的准备。在切换文件之前，记得保存index.html。

6.2　第一个脚本

首先要创建一个scripts文件夹和main.js文件。回忆一下如何使用Atom编辑器新建文件夹：按下Control键，右击面板左侧的ottergram，然后点击弹出菜单中的New Folder，最后在出现的提示框中输入scripts。

接着按下Control键，右击面板左侧的scripts并选择New File。在提示框中会有预输入的scripts/字样，在其后输入main.js，按下回车。

确保当前文件夹结构如图6-3所示。

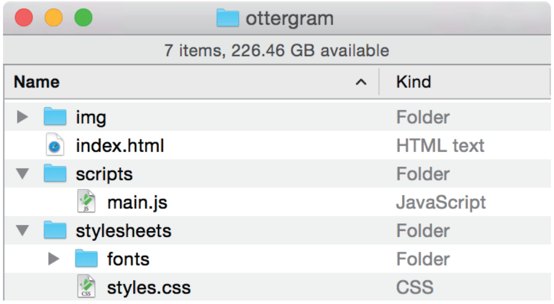


图6-3　ottergram文件夹结构

main.js这个名字对浏览器来说没有什么特殊的含义，不过这是很多前端开发者约定俗成的命名习惯。

切换到JavaScript之前的最后一件事就是启动browser-sync，但要先对之前使用过的命令进行一些改动：

```
browser-sync start --server --browser "Google Chrome"
                   --files "*.html, stylesheets/*.css, scripts/*.js"
```

在文件列表中加入路径scripts/*.js，使browser-sync除了能监听HTML和CSS之外，还能监听JavaScript的改动。

6.3　Ottergram 中的 JavaScript 描述

先制订计划后编码是一个很好的习惯。下面是需要在Ottergram中做的事情的简单描述。

(1) 获取所有的缩略图。

(2) 监控每个缩略图的点击事件。

(3) 如果发生了点击，根据缩略图的信息更新大图。

可以将第3点再拆分为3个步骤。

(1) 从缩略图的数据属性中获取图像的URL。

(2) 从缩略图的数据属性中获取标题文本。

(3) 将图像和标题设置到大图上。

下面是计划的图解（如图6-4所示）。

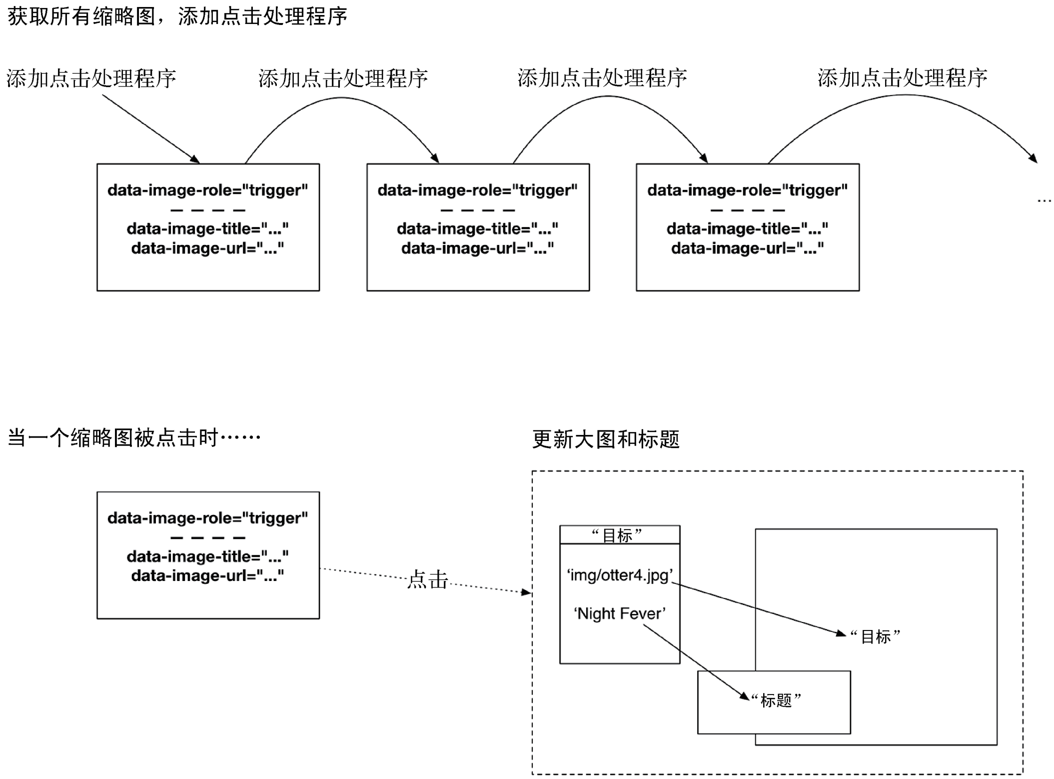


图6-4　Ottergram交互计划

本章将从最后一步开始讲述如何编写代码，这是一种“自底向上”的方法，在JavaScript中十分奏效。

6.4　声明字符串变量

首先为每个添加到标记中的数据属性创建字符串变量。（不熟悉这些术语也不用担心，马上会解释。）

在 main.js 的开头添加一个名为 `DETAIL_IMAGE_SELECTOR` 的变量，并且赋值为 `'[data-image-role="target"]'`。

```
var DETAIL_IMAGE_SELECTOR = '[data-image-role="target"]';
```

这部分代码虽然不长，但是值得仔细看看。从中间的=符号看起，这是一个**赋值操作符**。与数学中不同，等号在JavaScript中不意味着两个事物相等，而表示“将等号右边的值赋给等号左边的名称”。

在表达式中，等号的右边是一个文本**字符串**`'[data-image-role="target"]'`。字符串是

指一个由单引号分隔的表示文本的字符序列。单引号中的文本就是大图的属性选择器，这个字符串就是用来访问相应元素的线索。

表达式左边是一个**变量声明**。可以把变量想象成用来指定某个值的一个标签，这个值可以是数值，也可以是字符串（如本例）或者其他类型的值。使用`var`关键字可以创建一个叫作`DETAIL_IMAGE_SELECTOR`的变量。

接下来，在main.js中为大图标题选择器和缩略图锚选择器声明变量。同样把字符串赋值给这些选择器。

```
var DETAIL_IMAGE_SELECTOR = '[data-image-role="target"]';
var DETAIL_TITLE_SELECTOR = '[data-image-role="title"]';
var THUMBNAIL_LINK_SELECTOR = '[data-image-role="trigger"]';
```

所谓**变量**，顾名思义就是可以对其进行重新赋值的量，即值是可变的。开发者有时会使用全大写的变量名来表示这个变量的值不应该被改动，这是一个约定；其他语言使用**常量**来声明不可改变的名称。JavaScript正在转型：ES5没有常量，而ES6有。不过就像前文提到的，ES6还没有得到广泛支持。在常量得到良好支持之前，可以先遵循当前约定来标识一个不应当被改变的值。

顺便说下，字符串既可以由单引号分隔，也可以由双引号分隔。你可以任选一种，但本书使用单引号，所以也建议你至少在本书的项目中遵从这个约定。

6

如果你想要使用双引号，那么为了确保浏览器可以正确地解析所有代码，必须**转义**字符串中的所有双引号。要转义一个字符，需要像这样在字符前面加一个反斜杠：

```
var DETAIL_IMAGE_SELECTOR = "[data-image-role=\"target\"]";
```

使用单引号也不能完全保证你不需要转义字符。如果一个字符串由单引号分隔，其中又包含单引号或者撇号，则同样需要转义它们。

保存main.js。有了这些变量，就可以在Chrome开发者工具中玩转它们了。

6.5 操作控制台

开发者工具中非常有用的一部分就是控制台，你可以在里面输入JavaScript代码并且立即执行，这对迭代式地编写改变页面的JavaScript代码尤其有帮助。

在开发者工具中，点击Console标签，它位于Elements标签右侧（如图6-5所示）。

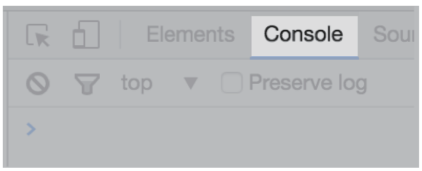


图6-5　选择Console标签

控制台有一个输入代码行的提示框，点击 > 符号的右侧使控制台进入待输入状态（如图6-6所示）。

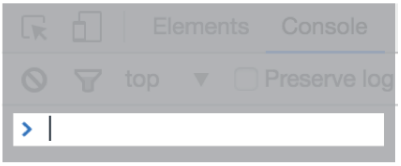


图6-6　控制台，待输入

在控制台中输入如下数学表达式：

```
137 + 349
```

按回车键，控制台会打印出结果（如图6-7所示）。

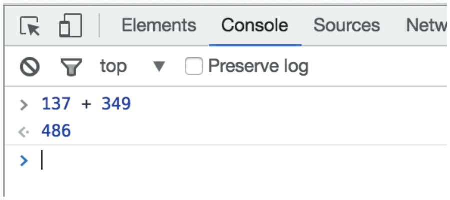


图6-7　计算数学表达式

简单来说，控制台的主要工作就是给出输入代码的值。

代码的顺序很重要。如果需要某些项作为一个组合来求值，可以用小括号括住它们。（这比不用括号而是依赖于JavaScript的优先级要简单多了。）在控制台中输入下面的表达式：

```
3 * ( (2 * 4) + (3 + 5) )
```

按回车键，控制台会按照正确的顺序进行运算（如图6-8所示）。（顺便说下，尽管为了保证可读性，在示例的数字和操作符间加入了空格，但你也可以不加，这对控制台没有影响。）

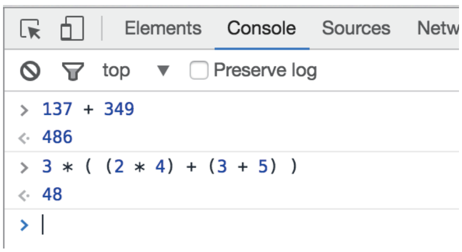


图6-8　计算更复杂的数学表达式

现在该使用之前声明的变量了。可以通过点击控制台面板左上角的⊘图标，也可以使用快捷键Command + K（Cntrol + K）来清空控制台中的内容。

键入`DETAIL_IMAGE_SELECTOR`。当键入前几个字母的时候，可以看到控制台已经提供了之前创建的变量的自动补全建议了（如图6-9所示）。

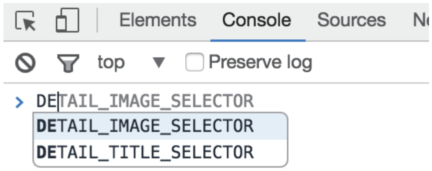


图6-9 控制台自动补全菜单

按下Tab键来选择控制台提供的自动补全。按下回车键，控制台显示`DETAIL_IMAGE_SELECTOR`的值为字符串`"[data-image-role="target"]"`。

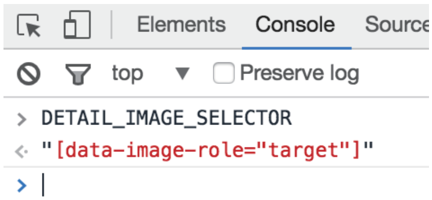


图6-10 控制台打印变量的值

（尽管main.js中使用的是单引号，但控制台打印的字符串通常都使用双引号。）

字符串是JavaScript的五个**基本**数据类型之一。（数值和布尔值也是其中的两个。）“基本”意味着它们代表的都是简单值，这个简单是相对于接下来将学到的JavaScript复杂类型而言的。

6.6 访问 DOM 元素

如上所见，控制台可以访问之前创建的变量。在更前面一点，我们说过要用这些变量来访问页面元素。现在可以试一下，在控制台中输入如下代码：

```
document.querySelector(DETAIL_IMAGE_SELECTOR);
```

按下回车键将显示大图的HTML。将鼠标悬停在控制台中的HTML信息上，可以看到页面中的大图是高亮的，与点击Elements面板中的HTML效果相同（如图6-11所示）。

在控制台敲入的这行代码中，`document`是浏览器内置的变量，可以通过它来访问网页。它的值并不是一个基本类型，而是一个复杂类型——**对象**。

`document`对象对应整个页面，它为获取页面元素的引用提供了一系列**方法**。方法是函数的一种（是被明确指定了所有者的函数，不过你现在不需要在意这个细节），是浏览器执行操作的步骤。而刚刚在控制台中输入的代码使用的就是`querySelector`方法。`document.querySelector`中的**点操作符**（或者说是句点）表示要访问对象的方法。

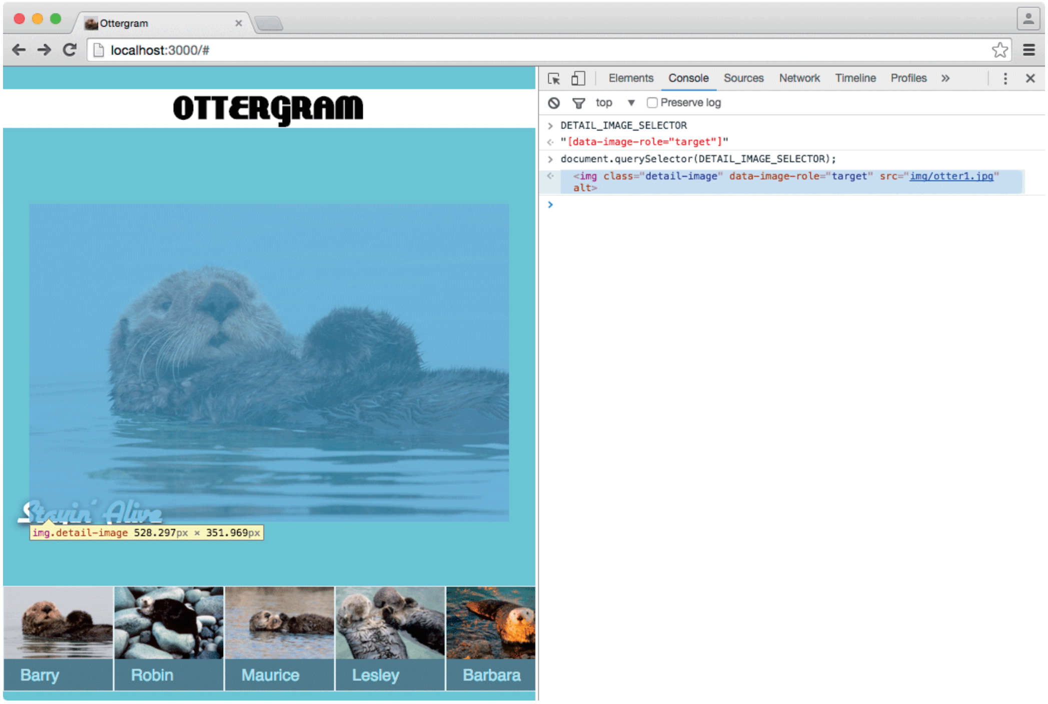


图6-11　控制台中的HTML与页面元素相对应

这里请求document使用它的querySelector方法找到匹配字符串'[data-image-role="target"]'的元素。querySelector返回一个找到的大图元素的引用（如图6-12所示）。

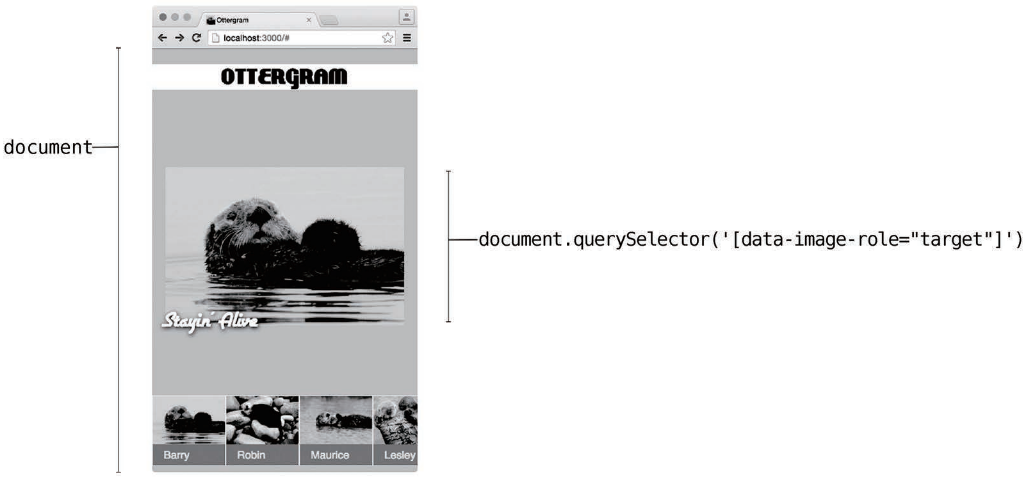


图6-12　访问由document和document.querySelector提供的页面

来说点术语吧。我们并没有真“请求”页面去匹配元素，只是**调用**了`document`的`query Selector`方法并且**传递**给它一个字符串。这个方法**返回**了一个大图元素的引用。

调用一个方法可以让它执行预先设定好的任务。大多数时候，需要给方法传递一些与任务相关的参数，参数要写在方法名后面的括号中。根据任务的不同，方法可能会返回一个可供使用的值。

之前`DETAIL_IMAGE_SELECTOR`已被赋值为`'[data-image-role="target"]'`，也就是说要把它传递给`querySelector`。

在这个场景下，`querySelector`使用这个字符串来搜索任何匹配此选择器的元素。搜索的时候，`document`不是真的在搜索整个页面，而是在搜索**文档对象模型**（Document Object Model，DOM）。DOM是浏览器对于一个HTML文档的内部表示，浏览器通过这种方式来读取和解释HTML。

在JavaScript中，可以通过`document`对象及`querySelector`这样的方法来与DOM交互。每个HTML标签在DOM中都有一个对应的元素，使用JavaScript可以与每个元素进行交互。（通常说到一个“元素”的时候，都是指一个“DOM元素”。）

再次在控制台中调用`document.querySelector`，传递`DETAIL_IMAGE_SELECTOR`来获取一个大图元素的引用，不过这次将引用赋值给一个名为`detailImage`的新变量：

```
var detailImage = document.querySelector(DETAIL_IMAGE_SELECTOR);
```

按下回车键，控制台会打印`undefined`（如图6-13所示）。不要慌，这不是个错误。

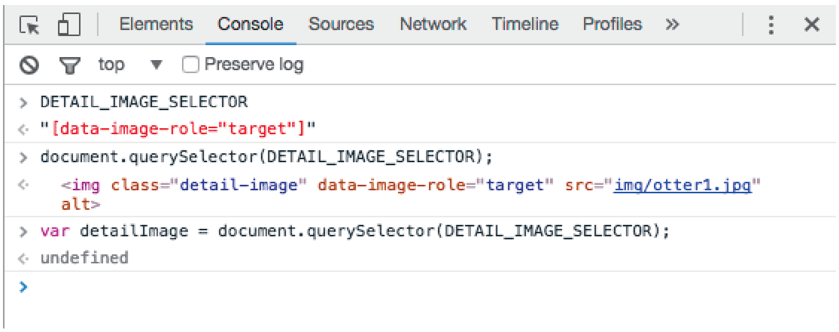


图6-13　在控制台中声明一个变量

控制台没做错，这里是在告诉你：声明一个变量并没有返回值。在JavaScript中，用`undefined`关键字来表示没有值。

但这不表示`detailImage`变量没有被赋值。想要检查这一点，只需在控制台中输入`detailImage`然后按下回车，就可以看到大图的HTML表示，显然与输入`document.querySelector(DETAIL_IMAGE_SELECTOR)`的效果是一样的（如图6-14所示）。

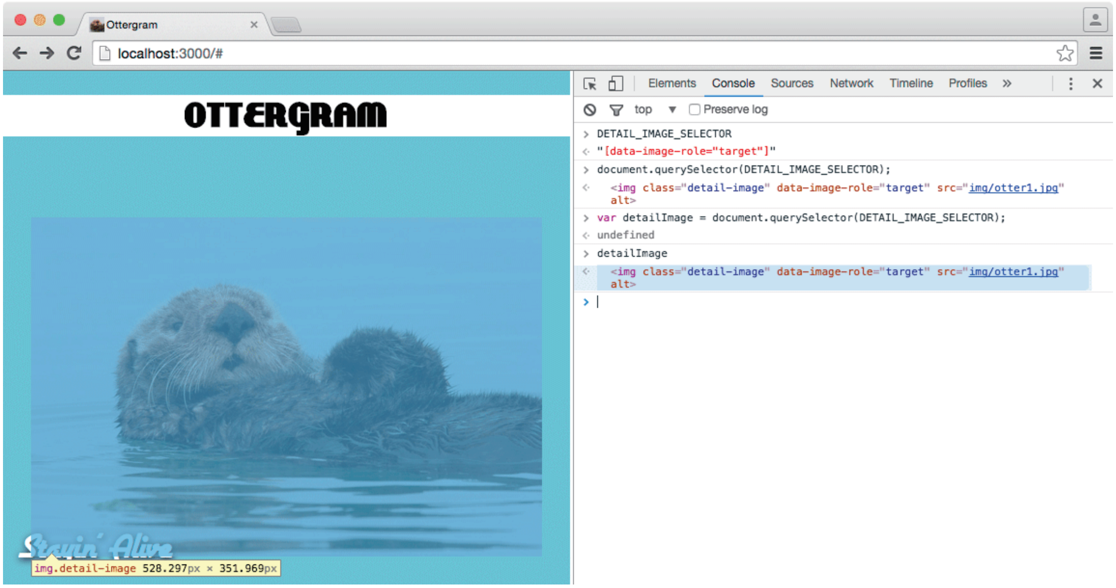


图6-14　检查detailImage的值

这样做有什么好处？通过把引用赋值给这个变量，可以在任何时候通过变量名使用被引用的元素。因此，不需要每次都输入document.querySelector(DETAIL_IMAGE_SELECTOR)，只输入detailImage就可以了。

获取到大图的引用之后，改变它的src属性就很轻松了。在控制台中，将detailImage.src赋值为'img/otter2.jpg'。

```
detailImage.src = 'img/otter2.jpg';
```

使用点操作符可以访问detailImage对象的src属性。属性和变量相似，只不过它属于一个特定的对象。将src赋值（或设置）为字符串'img/otter2.jpg'后，就可以看到另一只水獭出现在大图区域了（如图6-15所示）。

src属性对应index.html中<img>标签的src属性。鉴于这个关系，使用datailImage的setAttribute方法也可以得到同样的结果。

在控制台中调用这个方法并且传递两个字符串：属性名和新的值。

```
detailImage.setAttribute('src', 'img/otter3.jpg');
```

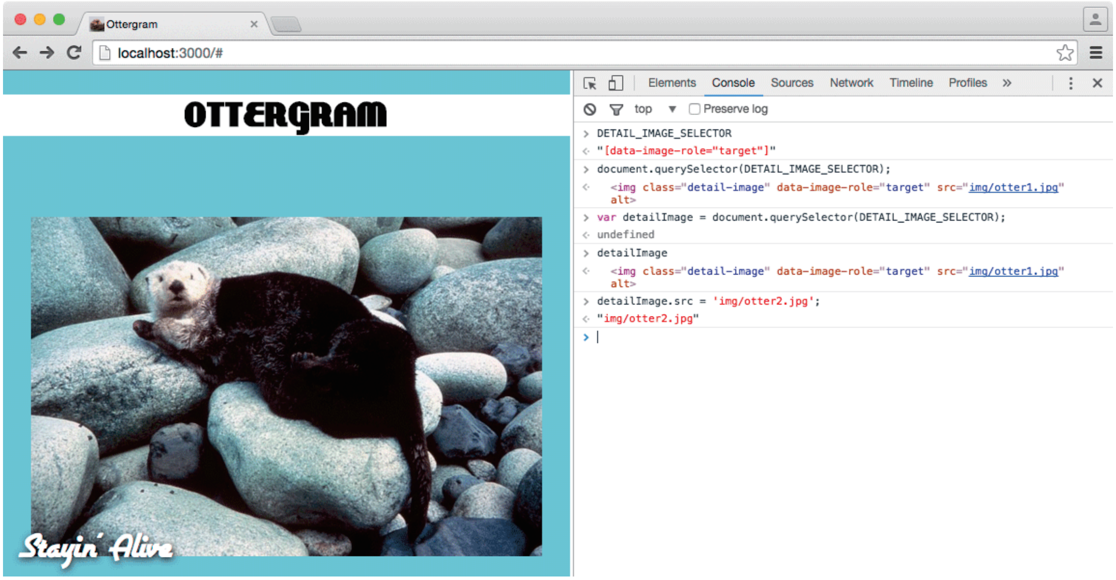


图6-15　为大图设置src属性

大图又一次改变了（如图6-16所示）。

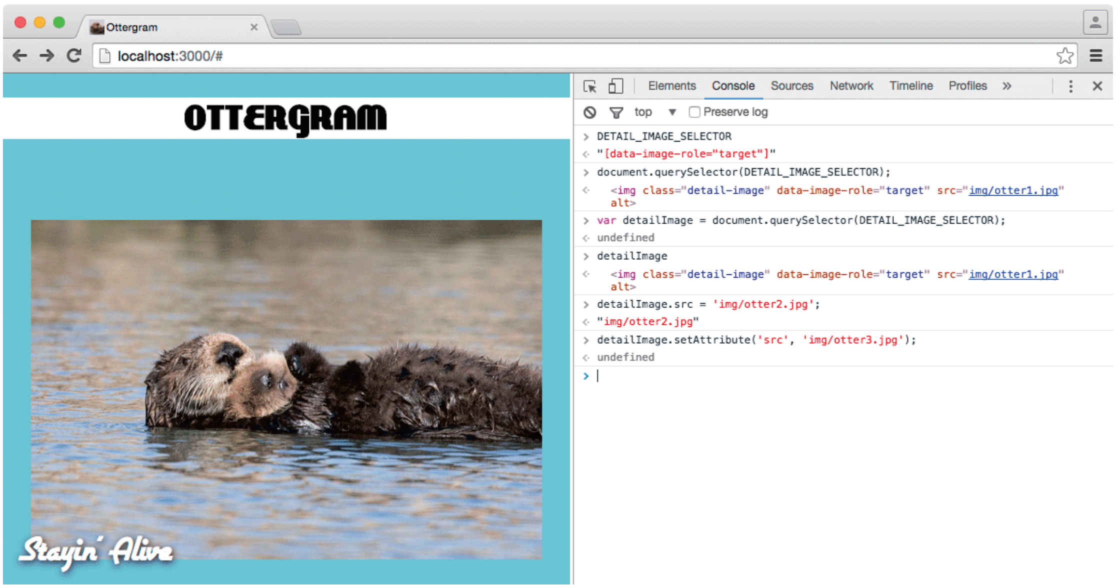


图6-16　使用setAttribute改变图片

所有要用来自动修改大图的代码都齐全了，接下来开始编写第一个JavaScript函数吧！

6

6.7　编写 setDetails 函数

前面我们已经接触了一些方法，知道调用它们可以让一段代码运行。函数和方法实际上就是一个可以复用的操作步骤列表。调用函数就像是说“做一个三明治”，而不是说“拿出两片面包，在一片上放上火腿、意大利腊肠、干酪，再把另一片面包放在最上面”。

本章会为Ottergram编写7个函数。第1个函数要完成两件事情：改变大图和大图标题。在main.js中添加如下**函数声明**。

```
var DETAIL_IMAGE_SELECTOR = '[data-image-role="target"]';
var DETAIL_TITLE_SELECTOR = '[data-image-role="title"]';
var THUMBNAIL_LINK_SELECTOR = '[data-image-role="trigger"]';

function setDetails() {
  'use strict';
  // 放置要运行的代码
}
```

上述代码使用function关键字声明了一个名为setDetails的函数。声明函数的时候，函数名后通常跟着一对圆括号，它们不是函数名的一部分，很快你就知道它们有什么用了。

圆括号后面是一对大括号，大括号内就是函数**体**。函数体中包含了函数要执行的操作，这些操作更正式的说法是**语句**。

函数的第一行是一个字符串'use strict';。在每个函数的开始使用这个字符串是告诉浏览器它符合JavaScript最新标准版的要求。（本章最后的延展阅读部分有更多关于严格模式的介绍。）

setDetails函数中的另一行是一条注释。和CSS注释一样，JavaScript注释也只是为开发者提供帮助，会被浏览器忽略。JavaScript的注释可以以//起始写在一行内；对于跨越多行的注释，可使用/* */的形式。这两种形式在JavaScript中都是正确的。

上一节我们已经在控制台中尝试过所有改变大图的语句了。再打开控制台，按上方向键，可以看到最近一条在提示框中输入过的语句。通过上、下方向键可以浏览语句的历史记录。

通过方向键，找到获取大图引用的语句var detailImage = document.querySelector(DETAIL_IMAGE_SELECTOR);。把这行从控制台中复制到main.js中，替换掉原来的注释，然后将控制台中调用detailImage.setAttribute方法的代码detailImage.setAttribute('src', 'img/otter3.jpg');复制粘贴过来。

main.js的setDetails函数应当像下面这样：

```
...
function setDetails() {
  'use strict';
  // 放置要运行的代码
  var detailImage = document.querySelector(DETAIL_IMAGE_SELECTOR);
  detailImage.setAttribute('src', 'img/otter3.jpg');
}
```

保存main.js，再回到控制台。输入下面的代码再按下回车来执行setDetails函数：

```
setDetails();
```

调用函数（即在函数名后面加上一对括号）会执行函数体中的代码。可以看到img/otter3.jpg现在展示为大图了（如图6-17所示）。

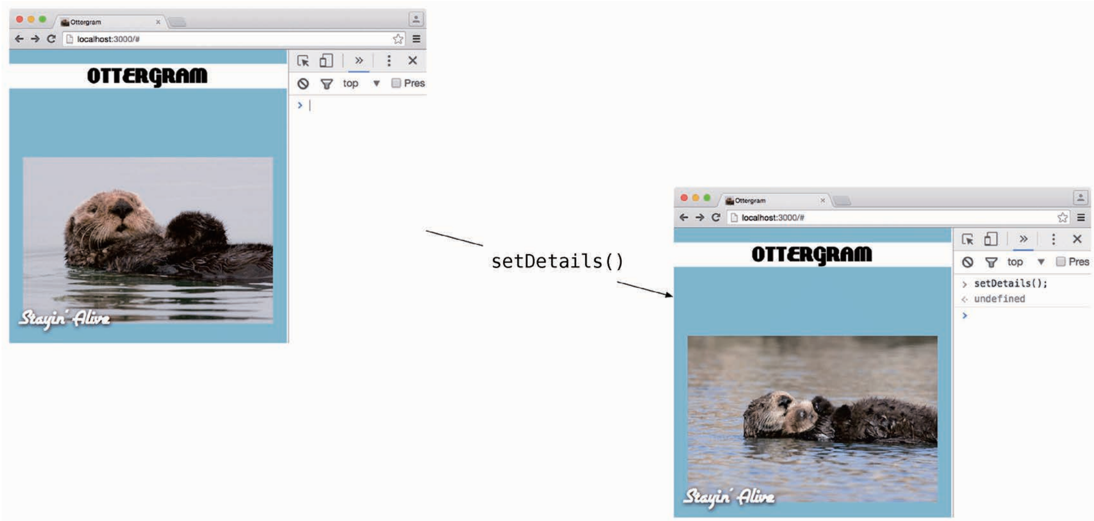


图6-17　执行setDetails来改变图片

setDetails改变了大图，但是没有改变大图标题。我们想它们都改变，所以当修改大图的时候，可以添加一条语句来获取元素的引用并且修改其属性。

在main.js的setDetails函数中，再一次调用document.querySelector，并且向其传递DETAIL_TITLE_SELECTOR参数。将结果赋给一个名为detailTitle的新变量，然后设置它的textContent属性为'You Should Be Dancing'。

```
...
function setDetails() {
  'use strict';
  var detailImage = document.querySelector(DETAIL_IMAGE_SELECTOR);
  detailImage.setAttribute('src', 'img/otter3.jpg');

  var detailTitle = document.querySelector(DETAIL_TITLE_SELECTOR);
  detailTitle.textContent = 'You Should Be Dancing';
}
```

textContent属性指的是一个元素内部的文本（不包括HTML标签）。

保存更改，再次在控制台中执行setDetails。现在大图和大图标题都改变了（如图6-18所示）。

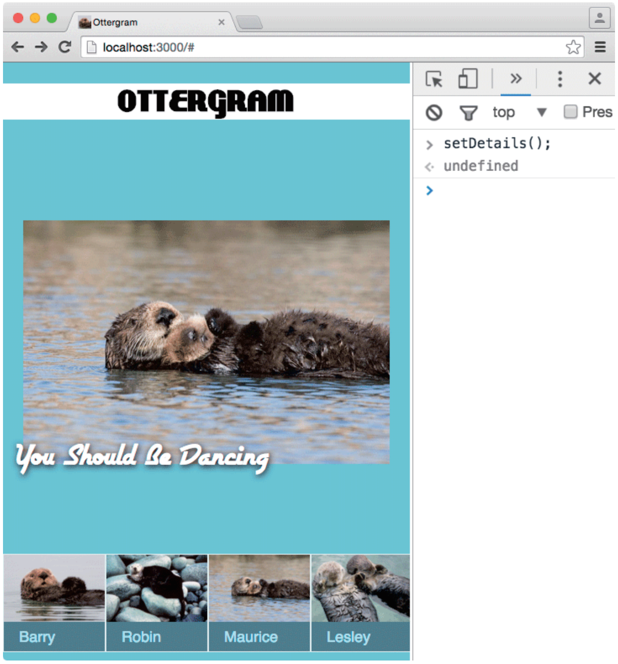


图6-18　使用setDetails修改图片和大图标题

通过形参声明接受实参

setDetails已经可以用于改变大图和大图标题了。然而每次运行它，它都将图片的src设置为img/otter3.jpg，将标题的textContent设置为'You Should Be Dancing'。如果想换成其他图片和标题呢？

办法就是在调用setDetails的时候告诉它要使用的**图片**和**文字**。

为了达到这个目的，需要让函数可以接受**实参**——传递给函数的值。这样的话，就需要在函数声明的时候列举**形参**。

为main.js的setDetails添加两个形参：

```
...
function setDetails(imageUrl, titleText) {
  'use strict';
  var detailImage = document.querySelector(DETAIL_IMAGE_SELECTOR);
  detailImage.setAttribute('src', 'img/otter3.jpg');

  var detailTitle = document.querySelector(DETAIL_TITLE_SELECTOR);
  detailTitle.textContent = 'You Should Be Dancing';
}
```

然后使用这些形参来替换'img/otter3.jpg'和'You Should Be Dancing'：

```
...
function setDetails(imageUrl, titleText) {
  'use strict';
  var detailImage = document.querySelector(DETAIL_IMAGE_SELECTOR);
  detailImage.setAttribute('src', 'img/otter3.jpg');
  detailImage.setAttribute('src', imageUrl);

  var detailTitle = document.querySelector(DETAIL_TITLE_SELECTOR);
  detailTitle.textContent = 'You Should Be Dancing';
  detailTitle.textContent = titleText;
}
```

传递给`setDetails`的值被赋给了两个形参：`imageUrl`和`titleText`。保存main.js，然后在控制台中试一下能否正常运行。

调用`setDetails`并且向其传递`'img/otter4.jpg'`和`'Night Fever'`这两个值。（确保用逗号分隔这两个值。）

```
setDetails('img/otter4.jpg', 'Night Fever');
```

应该看到新的图片和标题文本，如图6-19所示。

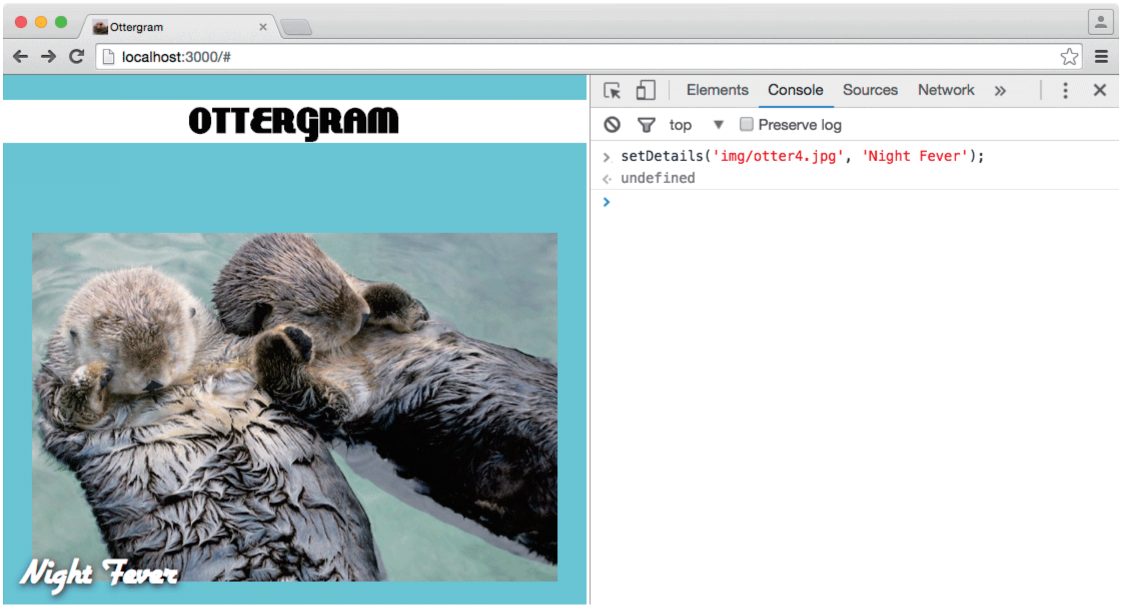


图6-19　给`setDetails`传值

形参和实参之间有一个很重要的区别。**形参是函数的一部分**。在JavaScript中，形参指的是声明在函数体中的变量。**实参则是调用函数时实际提供的值**。

还需要注意的是，无论实参使用了什么变量名，为了可以在函数体中使用，它们的值总是对应到形参上。例如，若使用了两个变量来保存图片URL和标题文本，那么当调用`setDetails`时，

可以将这两个变量作为实参传递。

```
var otterOneImage = 'img/otter1.jpg';
var otterOneTitle = 'Stayin\' Alive';

setDetails(otterOneImage, otterOneTitle);
```

setDetails接受值，再将它们赋值给imageUrl和titleText两个形参，然后运行函数体中的代码。代码中用到imageUrl和titleText,将它们作为实参传递给document.querySelector。

与变量名类似，形参就像值的一个标签。你可以使用任何你喜欢的形参名，不过更推荐使用语义化的名字，这样的代码可读性更高，也更容易维护。

6.8　从函数返回值

现在已经完成了计划中的第1项（更确切地说是最后1项），也在这个过程中学到了一些JavaScript技术。继续进行任务清单中接下来的两项内容：从缩略图中获取图片和标题。想要完成这些任务，需要编写一个新的函数。

在main.js中添加一个imageFromThumb的函数声明，它接受一个参数thumbnail作为缩略图锚元素的引用。它将检索并返回data-image-url属性的值。

```
...
function setDetails(imageUrl, titleText) {
  ...
}

function imageFromThumb(thumbnail) {
  'use strict';
  return thumbnail.getAttribute('data-image-url');
}
```

getAttribute方法与之前在setDetails函数中使用的setAttribute方法功能相反，它只需要一个参数——属性名。

与setDetails不同，imageFromThumb函数使用了return关键字。当调用一个含有return语句的函数时，它会返回一个值。querySelector就是这样的函数。当调用它的时候，它会返回一个值，你可以将这个值赋给一个变量。

保存main.js，然后在控制台中试一下下面的代码，代码行之间要输入回车。

```
var firstThumbnail = document.querySelector(THUMBNAIL_LINK_SELECTOR);
imageFromThumb(firstThumbnail);
```

控制台显示返回的值是一个字符串"img/otter1.jpg"，这是因为imageFromThumb返回了缩略图的data-image-url。

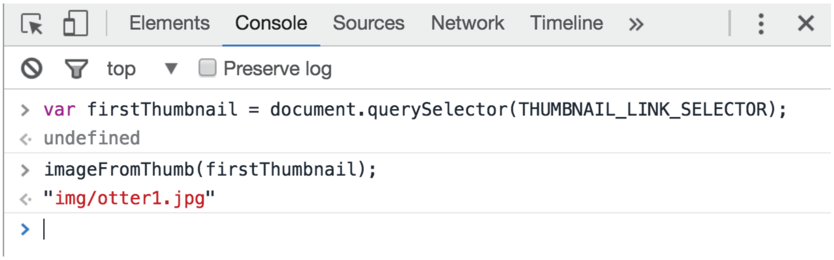


图6-20　imageFromThumb的返回值

需要注意的是，所有位于return语句之后的语句都不会执行，return语句可以有效地停止正在运行的函数。

接下来要编写的函数会接受一个缩略图元素的引用，然后返回标题文本。

在main.js中添加一个titleFromThumb的函数声明，其中包含一个thumbnail参数。这个函数将返回data-image-title属性的值。

```
...
function imageFromThumb(thumbnail) {
  ...
}

function titleFromThumb(thumbnail) {
  'use strict';
  return thumbnail.getAttribute('data-image-title');
}
```

保存main.js并且在控制台中进行试验：

```
var firstThumbnail = document.querySelector(THUMBNAIL_LINK_SELECTOR);
titleFromThumb(firstThumbnail);
```

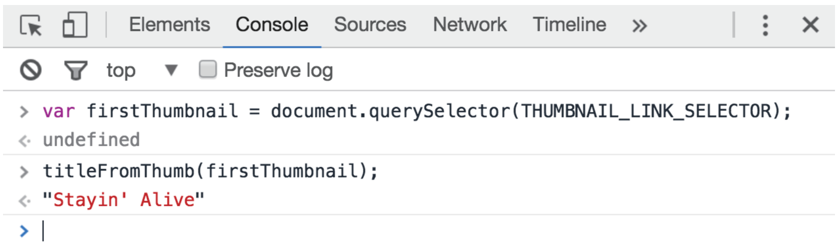


图6-21　titleFromThumb的返回值

接下来要写的函数是为之前3个函数服务的，以便于不用分别调用它们。这个函数接受一个缩略图元素的引用，然后调用setDetails，并且传递从imageFromThumb和titleFromThumb得到的返回值。

在main.js中添加setDetailsFromThumb。

```
...
function titleFromThumb(thumbnail) {
  ...
}

function setDetailsFromThumb(thumbnail) {
  'use strict';
  setDetails(imageFromThumb(thumbnail), titleFromThumb(thumbnail));
}
```

请注意，调用`setDetails`时传递了两个参数，这两个参数都是函数调用。这段代码执行的过程是怎么样的呢？

在`setDetails`真正被调用前，它的参数会先还原为最简单的值。首先运行`imageFromThumb(thumbnail)`并返回一个值，接着运行`titleFromThumb(thumbnail)`并返回一个值，最后`setDetails`被调用的时候，传递从`imageFromThumb(thumbnail)`和`titleFromThum(thumbnail)`返回的两个值。图6-22展示了这个过程。

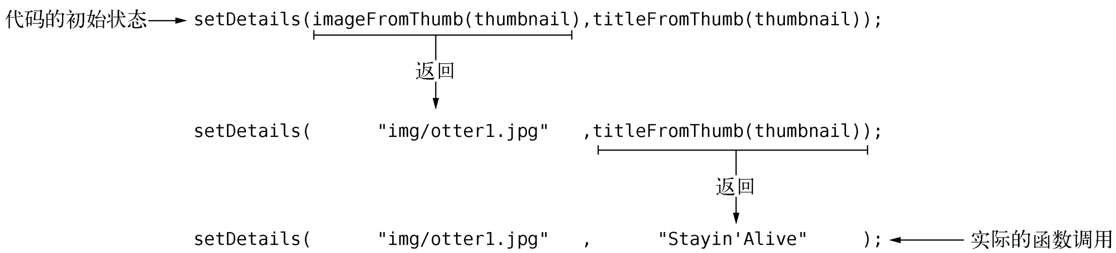


图6-22　将函数调用作为参数

保存main.js，目前已经完成了通过缩略图检索数据属性值并使用这些值更新大图和标题的代码。

结束了低层次的操作之后，接下来要编写当用户点击缩略图时将缩略图的信息传到大图的代码。

6.9　添加事件监听器

浏览器是个很忙碌的软件，它会注意到每一次触碰、点击、滚动以及按键，这些行为都是浏览器可能要响应的**事件**。为了使网站动态化并且可交互，可以在这些事件发生的时候触发某些代码。本节将会为每个缩略图添加**事件监听器**。

事件监听器是一个对象。正如其名，它可以“监听”某个特定事件（如鼠标点击）。当指定事件发生时，事件监听器就会触发一个函数调用以响应事件。

（像单击和双击这样的鼠标事件，还有按键等键盘事件都是最常见的事件类型。MDN上有完整的事件列表，请见developer.mozilla.org/en-US/docs/Web/Events。）

包括`document`在内的每个DOM元素都有`addEventListener`方法。像之前一样，先在控制

台中试验代码，然后将测试过的代码作为函数写进main.js中。

切换到Chrome，在控制台中输入如下代码，需要使用Shift + Enter来输入换行符。代码输入完毕后按下回车键。

```
document.addEventListener('click', function () {
  console.log('you clicked!');
});
```

刚刚的代码为document对象添加了一个监听当前页面所有点击事件的监听器。当发现点击事件时，事件监听器就会使用内置的console.log方法在控制台中打印“you clicked!”（如图6-23所示）。

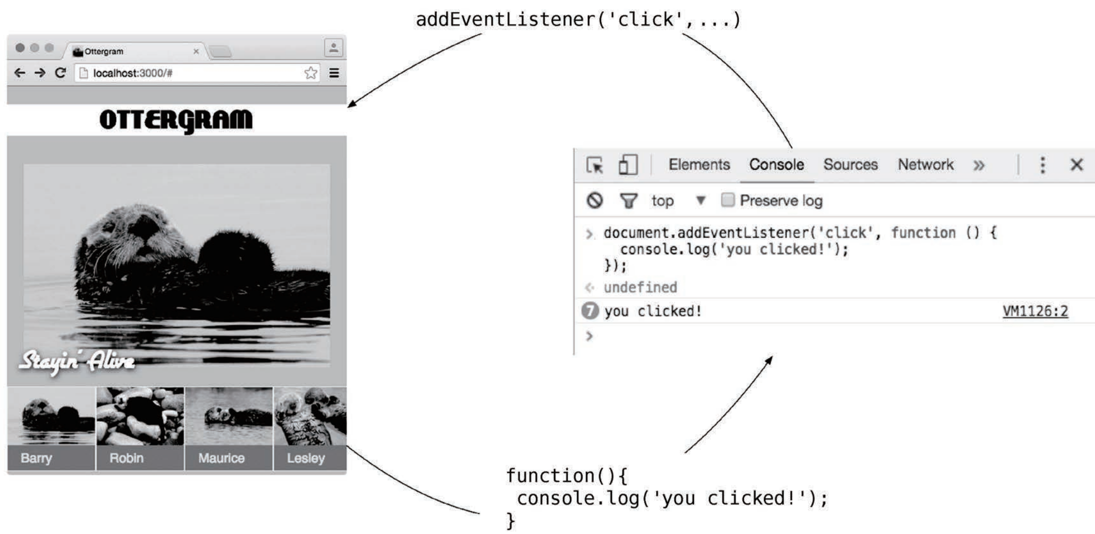


图6-23　添加点击事件的监听器

点击页头、大图或者背景都可以看到控制台中打印了文本“you clicked!”。（不要点击缩略图，那样会导致离开Ottergram的index.html页面。如果不在index.html页面上，浏览器就不会加载和运行你写的所有标记、CSS和JavaScript。）

addEventListener接受两个参数：一个表示事件名的字符串和一个函数。一旦事件发生在当前元素上，addEventListener就会运行这个函数。这种编写函数的方式乍一看有些奇怪，它叫**匿名函数**。

到现在为止，我们使用过setDetails和titleFromThumb这样的**命名函数**。显然，命名函数有名字（这不奇怪），并且是通过函数声明创建的。

你也可以编写函数**字面量**，就像编写42这样的数值字面量和"Barry the Otter"这样的字符串字面量一样。函数字面量的另一个名字就是**匿名函数**。

匿名函数通常作为其他函数的参数来使用，如传递给document.addEventListener的第二

个参数。将一个函数传递给另一个函数的做法在JavaScript中十分常见，并被称为**回调模式**，因为作为参数传递的这个函数通常在未来的某个时间点会被“调用回来”。

当然，使用一个命名函数作为回调函数也没问题，不过多数前端开发者会选择匿名函数，因为它比命名函数更加灵活。我们马上会介绍回调函数的原理。

现在，要为一个缩略图添加事件监听器。在控制台中输入如下代码（使用Shift + Enter来为调用`firstThumbnail.addEventListener`那里添加换行符）：

```
var firstThumbnail = document.querySelector(THUMBNAIL_LINK_SELECTOR);
firstThumbnail.addEventListener('click', function () {
  console.log('you clicked!');
});
```

这时试着点击第一个缩略图（水獭Barry，最左边那个），浏览器会打开水獭Barry的大图。发生了什么？要知道，每个缩略图都被包裹在一个锚标签中，而标签的`href`属性指向了图片，如img/otter1.jpg。用户点击链接时，浏览器的默认行为就是打开`href`属性指向的文件。

不过我们并不希望点击缩略图时离开Ottergram，因此不得不对锚标签再做些改动。好在此时可以通过回调函数来解决这个问题。

本章前面曾提到，函数执行任务时我们不用考虑其内部实现。通常，我们只需要知道传什么参数，以及函数返回什么信息就行了。当把一个回调函数作为参数传递时，就多了一件需要知道的事情：什么信息会被传递给回调函数。

调用`addEventListener`就是在告诉浏览器：“当`firstThumbnail`被点击的时候，调用这个函数”，接下来浏览器就会等待着这个元素被点击。如果点击发生了，浏览器就会记录下事件的所有细节（例如鼠标的准确位置、左键点击还是右键点击、单击还是双击等）。接着，浏览器就会把包含这些信息的一个对象传递给函数，这个对象叫作**事件对象**。

其中的关系如图6-24所示，其中使用了`addEventListener`的组合实现。



图6-24　传递一个需要参数的匿名函数

接下来像之前一样向addEventListener传递一个匿名函数，不过这次的匿名函数需要接受一个参数。确保Ottergram在index.html页面并在控制台中输入：

```
var firstThumbnail = document.querySelector(THUMBNAIL_LINK_SELECTOR);
firstThumbnail.addEventListener('click', function (event) {
  event.preventDefault();
  console.log('you clicked!');
  console.log(event);
});
```

每当firstThumbnail被点击时，浏览器都会调用这个匿名函数，并且将事件对象传递给匿名函数。通过这个对象（即event），可以调用它的preventDefault方法，这个方法可以阻止链接让浏览器跳到另一个页面。最后，调用console.log并传入event对象，以便在开发者工具中看看它到底包含哪些信息。

现在点击第一个缩略图，浏览器会停在Ottergram页面，而事件被记录到了控制台中：MouseEvent {isTrusted: true}。点击MouseEvent旁边的详情箭头可以看到事件的更多信息（如图6-25所示），比如鼠标在页面上的坐标，哪个鼠标按键被点击，以及是否有任何特殊按键在点击期间被按下。

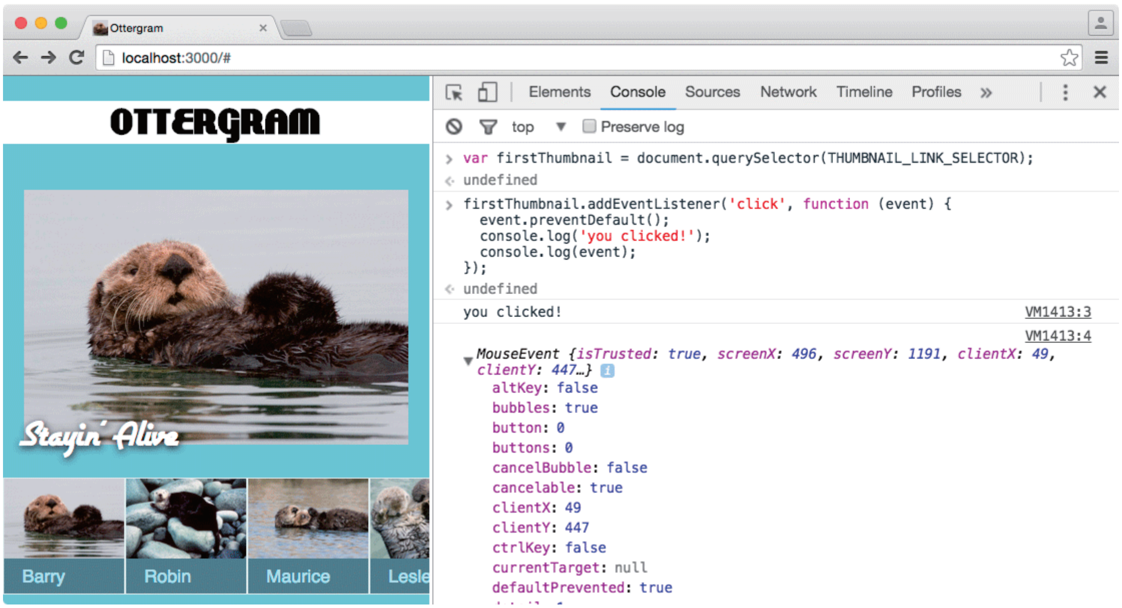


图6-25　阻止默认事件并且打印事件对象

现在不必关注事件对象中的各种属性，只需要清楚它包含了许多有关已触发事件的信息即可。

另外，回调函数的参数命名不一定是event。无论叫什么名字，它都会自动与传递的值进行匹配。因此可以使用任何你喜欢的名字，但最好的做法是使用描述性的名字（就像之前做的那样），以提高代码的可读性和可维护性。

现在已经有一个接受缩略图并为其添加事件监听器的函数了。接下来在main.js中添加addThumbClickHandler的函数声明，它应当定义一个名为thumb的参数。

可以把控制台中试验过的addEventListener代码复制到addThumbClickHandler函数体中。为了调用thumb.addEventListener，需要对其做些修改。但现在，只需在回调函数中调用event.preventDefault即可。

```
...
function setDetailsFromThumb(thumbnail) {
  ...
}

function addThumbClickHandler(thumb) {
  'use strict';
  thumb.addEventListener('click', function (event) {
    event.preventDefault();
  });
}
```

回调函数作为addThumbClickHandler的一部分，可以访问已声明的参数thumb，并且把它传递给setDetailsFromThumb的调用。

```
...
function addThumbClickHandler(thumb) {
  'use strict';
  thumb.addEventListener('click', function (event) {
    event.preventDefault();
    setDetailsFromThumb(thumb);
  });
}
```

和其他编程语言一样，JavaScript同样也有定义及访问变量和函数的规则。被传递给addEventListener的匿名函数可以访问setDetailsFromThumb是因为setDetailsFromThumb声明在**全局作用域**中，这意味着它可以被任何函数以及控制台访问。DETAIL_IMAGE_SELECTOR等变量也是如此，它同样声明在全局作用域中。

然而，在setDetails中声明的变量detailImage和detailTitle只能在setDetails函数体中被访问，无法通过控制台或者其他函数来访问，因为这些变量被定义在了setDetails的**函数作用域**（也被称为**局部作用域**）中。一个函数的参数的行为与声明在函数中的变量非常相似，它们也是该函数作用域中的一部分。

正常来讲，函数是不能访问其他函数作用域中的变量和参数的。但addThumbClickHandler函数定义了一个可以被其他函数（也就是传递给addEventListener函数的回调函数）访问的thumb参数。因为该回调函数位于addThumbClickHandler函数的作用域中，所以这一切才可能发生。

本章最后的“延展阅读”有关于这一过程的更多内容。

6.10 访问所有缩略图

我们已经在控制台中为第一张缩略图添加了事件监听器。现在，为给所有缩略图添加事件监听器，使用一个新的DOM方法。

在检索大图和大图标题时，可以使用`document.querySelector`方法在DOM中搜索与传入的选择器相匹配的元素。`document.querySelector`只返回一个值，即使传入的选择器可以匹配多个元素，它也只返回一个值。

而`document.querySelectorAll`方法将返回所有匹配的元素。在控制台中调用`document.querySelectorAll(THUMBNAIL_LINK_SELECTOR)`，将可以看到一个包含锚元素的列表（如图6-26所示）。

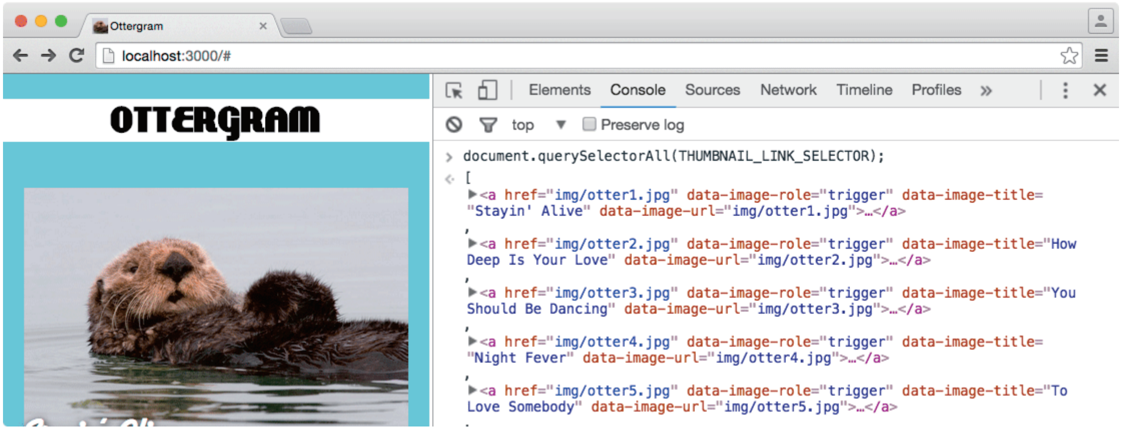


图6-26　`document.querySelectorAll`返回多个相匹配的结果

知道了这些之后，就可以更好地测试`setDetailsFromThumb`函数了。在控制台中，将调用`document.querySelectorAll(THUMBNAIL_LINK_SELECTOR)`的结果赋值给一个名为`thumbnails`的变量。使用**方括号**来检索`thumbnails`列表中的第5个元素，并将其传入`setDetailsFromThumb`函数中。方括号中的数值可以作为下标来指定一个元素，下标从0开始，因此第5项下标为4。

在控制台中输入如下代码：

```
var thumbnails = document.querySelectorAll(THUMBNAIL_LINK_SELECTOR);
setDetailsFromThumb(thumbnails[4]);
```

在控制台中执行上述代码后，`thumbnails`列表中的一项便传给了`setDetailsFromThumb`函数，并成功改变了大图和标题（如图6-27所示）。

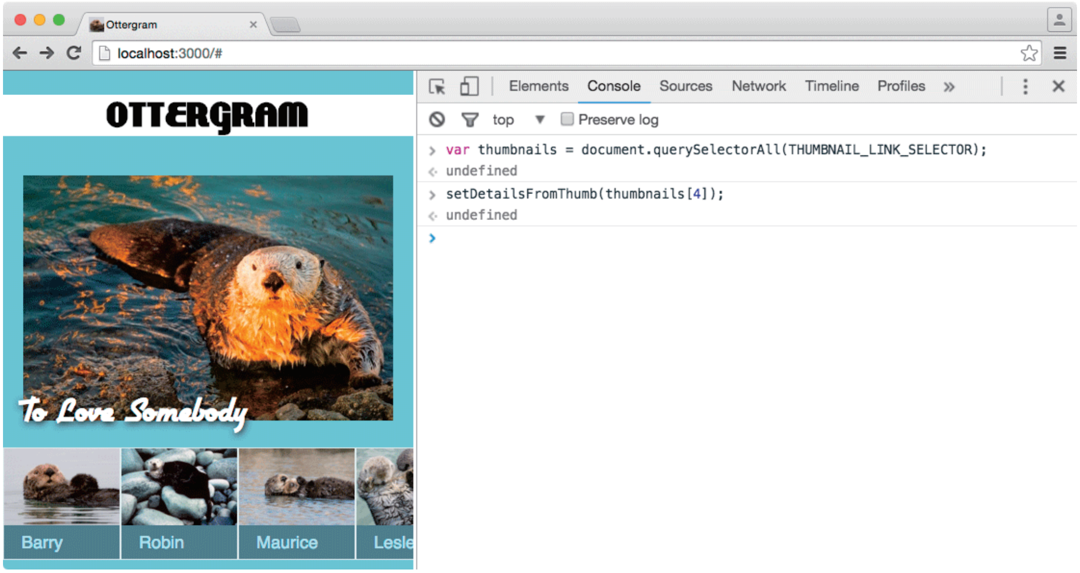


图6-27　从querySelectorAll向setDetailsFromThumb传入一项

在main.js中添加一个名为getThumbnailsArray的函数，并把检索匹配THUMBNAIL_LINK_SELECTOR元素的结果和赋值给thumbnails变量的代码粘贴进去。

```
...
function addThumbClickHandler(thumb) {
  ...
}

function getThumbnailsArray() {
  'use strict';
  var thumbnails = document.querySelectorAll(THUMBNAIL_LINK_SELECTOR);
}
```

在使用DOM方法时还要注意一个问题。刚刚的DOM方法返回的是一个元素列表而不是**数组**，具体来说，返回的是一个**节点列表**。数组和节点列表都是一些项的列表，不同的是数组拥有一系列功能强大的方法用于处理这些集合，而在Ottergram中同样需要使用其中的某些方法。

因此，有必要通过一种方式将querySelectorAll返回的节点列表转换为一个数组，虽然这种方式看起来有些奇怪。这种语法可以暂且不深究，这是一种将节点列表转换为数组的向后兼容的方式。在main.js中进行如下修改：

```
...
function getThumbnailsArray() {
  'use strict';
  var thumbnails = document.querySelectorAll(THUMBNAIL_LINK_SELECTOR);
  var thumbnailArray = [].slice.call(thumbnails);
  return thumbnailArray;
}
```

现在已经获取到所有的水獭缩略图了。试着将它们与之前的事件监听代码关联起来，让大图和标题响应点击，进行改变。

6.11 迭代缩略图数组

将缩略图与事件处理代码关联起来是项简单的工作，只要编写一个作为整个Ottergram项目逻辑起点的函数就好了。其他编程语言有内置的启动应用的机制，而JavaScript没有。但是不用担心，这实现起来也很容易。

先在main.js结尾添加一个名为`initializeEvents`的函数，该方法会把所有步骤联系起来，从而使Ottergram具有交互性。它首先会获取缩略图数组，然后遍历整个数组，给其中每个元素添加点击事件处理程序。写完这个函数后，需要在main.js的最后添加一条调用`initializeEvents`函数的语句来执行它。

在新函数的函数体中添加一条调用`getThumbnailsArray`函数的语句并将结果（缩略图数组）赋给一个名为`thumbnails`的变量。

```
...
function getThumbnailsArray() {
  ...
}

function initializeEvents() {
  'use strict';
  var thumbnails = getThumbnailsArray();
}
```

接下来，需要一项一项地检查整个缩略图数组。当访问每个数组元素时，调用`addThumbClickHandler`函数并将缩略图元素传给它。这看起来需要很多步骤才能完成，不过因为`thumbnails`是一个数组，因此可以通过一个简单的方法调用来完成上述所有操作。

在main.js中加入调用`thumbnails.forEach`函数的语句，并将`addThumbClickHandler`作为回调函数传进去。

```
...
function initializeEvents() {
  'use strict';
  var thumbnails = getThumbnailsArray();
  thumbnails.forEach(addThumbClickHandler);
}
```

请注意，上面的代码中传递了一个命名函数作为回调。后面将会提到，这不总是一个好办法。不过在目前这个例子中这样做并不会引发错误，因为`addThumbClickHandler`函数只需要在`forEach`函数调用它时传递给它的信息，即`thumbnails`数组中的一个元素。

最后，在main.js的末尾处添加调用`initializeEvents`函数的语句来触发上述所有操作。

```
...
function initializeEvents() {
  'use strict';
```

```
    var thumbnails = getThumbnailsArray();
    thumbnails.forEach(addThumbClickHandler);
  }

  initializeEvents();
```

记住，浏览器会一边逐行读入JavaScript代码，一边按序执行它们。对于main.js中的大部分代码而言，浏览器仅执行变量和函数的声明，而当其读到最底端的`initializeEvents();`时，它将执行该函数。

保存修改并返回到浏览器中，单击几张不同的缩略图来欣赏一下刚刚的劳动成果（如图6-28所示）。

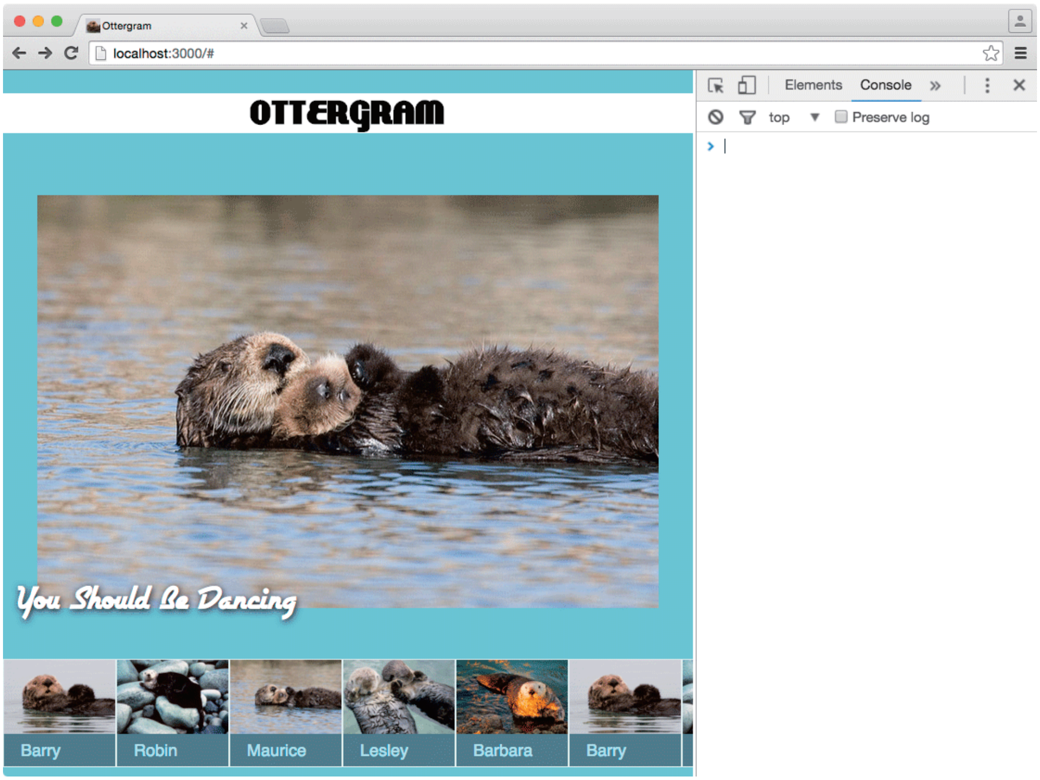


图6-28　你真的可以跳个舞了

坐下来，点点水獭玩一玩。稍事休息，下面还有很多构建网站交互层的工作和知识等着你。下一章将通过添加额外的视觉效果来结束Ottergram项目。

6.12　中级挑战：劫持链接

Chrome开发者工具为实现当前访问的页面提供了很多帮助。接下来的这个挑战便是更改搜

索结果页上的所有链接来使得它们无法跳转到其他地方去。

访问最喜欢的搜索引擎并搜索“otters”，打开开发者工具并切换到控制台，参考Ottergram中所写的函数，为所有链接添加一个事件监听器并禁用它们响应单击事件的默认行为。

6.13 高级挑战：随机的水獭

编写一个函数，随机修改水獭缩略图的`data-image-url`，使得大图无法与缩略图相匹配。使用你自己选择的图片的URL（去搜索引擎上搜索“tacocat”应该能找到不错的结果）。

附加挑战：编写一个函数，将水獭缩略图的URL重设为最初的`data-image-url`值，然后再随机修改其中一个。

6.14 延展阅读：严格模式

什么是严格模式？它为何出现？严格模式最初作为JavaScript的一个更清洁的模式被创造出来，用以捕捉一些特定的代码错误（比如变量名输入错误），使开发者尽可能避免该语言中最易出错的部分，并且禁用了部分语言中通常并不好的特性。

严格模式有很多有优点。

- 强制使用`var`关键字。
- 不需要`with`语句。
- 在使用`eval`函数时加入了许多限制。
- 将函数的参数中出现重复名称的情况判定为语法错误。

要实现所有这些特性，仅需要在函数的前面加上`'use strict'`即可。这个命令的另一个好处是可被不支持该功能的旧式浏览器忽略（它们仅将该命令看作字符串）。

在MDN上（developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode）可以阅读更多关于严格模式的内容。

6.15 延展阅读：闭包

前文曾经提到过，比起命名函数，开发者通常更愿意使用匿名函数作为回调函数，`addThumbClickHandler`也说明了为何匿名函数是一个更好的选择。

假设现在将一个叫作`clickFunction`的命名函数作为回调函数。在这个函数中需要访问`event`对象，因为它会被传入`addEventListener`函数中，但是`clickFunction`的函数体无法访问`thumb`对象，该参数仅能在`addThumbClickHandler`函数中被访问：

```
function clickFunction (event) {
  event.preventDefault();

  setDetailsFromThumb(thumb); // <---这行代码会导致报错
}
```

6

```
function addThumbClickHandler(thumb) {
  thumb.addEventListener('click', clickFunction);
}
```

但使用匿名函数便可以使其访问thumb参数，因为它也在addThumbClickHandler的函数体内。当一个函数定义在另一个函数内时，前者可以使用后者所有的参数和变量。在计算机科学中，这种情况被称为**闭包**。

当addThumbClickHandler函数被执行时，它调用addEventListener，后者将单击事件与回调函数关联起来，在内部为回调函数维护了一个引用并且在事件发生时执行回调函数。

从技术上来讲，当回调函数最终执行时，addThumbClickHandler函数的变量和参数将不再存在，它们在addThumbClickHandler函数结束后便消失了。然而，回调函数“捕获”了addThumbClickHandler中的变量和参数的值，并使用捕获的值来执行语句。

如果想深入了解闭包，可以在MDN中阅读相关内容。

6.16 延展阅读：NodeList 对象和 HTMLCollection 对象

有两种检索DOM中元素的方法：第一种是使用document.querySelectorAll，它返回一个NodeList对象；另外一种方法是使用document.getElementsByTagName，它返回一个HTMLCollection对象。后者与前者的区别在于你只能将标签名作为字符串传入它，比如"div"或者"a"。

NodeList和HTMLCollection对象都不是真正的数组，它们缺少数组的一些方法，比如forEach，不过它们具有一些很有意思的属性。

HTMLCollection对象是**动态的结点**，这意味着在修改DOM时，不用再次调用document.getElementsByTagName，HTMLCollection的内容会自动变化。

可以在控制台中输入下面的语句来观察上述性质和行为。

```
var thumbnails = document.getElementsByTagName("a");
thumbnails.length;
```

获取了HTMLCollection中所有的锚元素后，在控制台中输出结果列表的length。

现在通过开发者工具中的Elements面板来删除页面中的一些锚标签：按住Control键，右击列表项，并在弹出菜单中选择Delete element（如图6-29所示）。

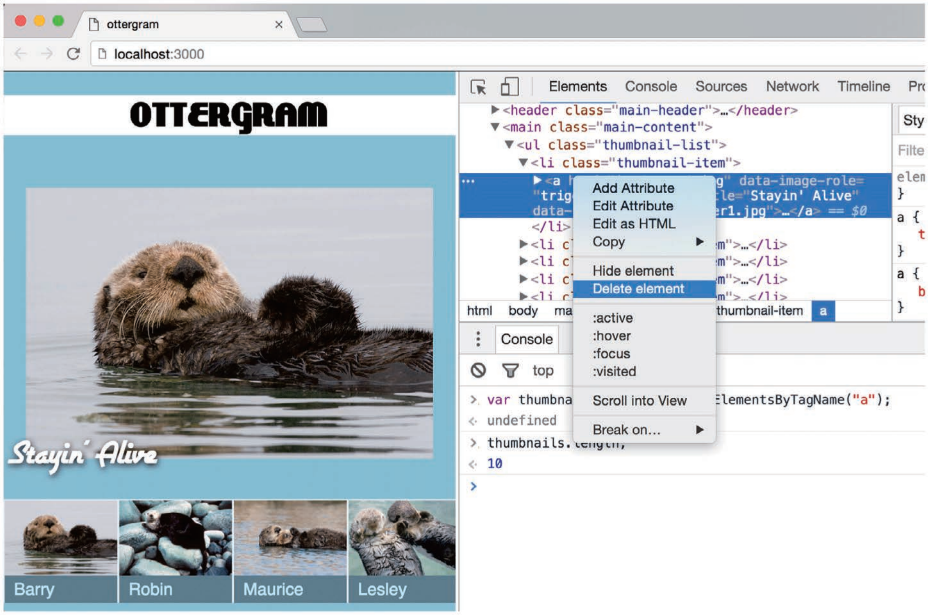


图6-29　使用开发者工具删除一个DOM元素

多删几个，然后在控制台中再次键入`thumbnails.length`，可以看到长度发生了变化（如图6-30所示）。

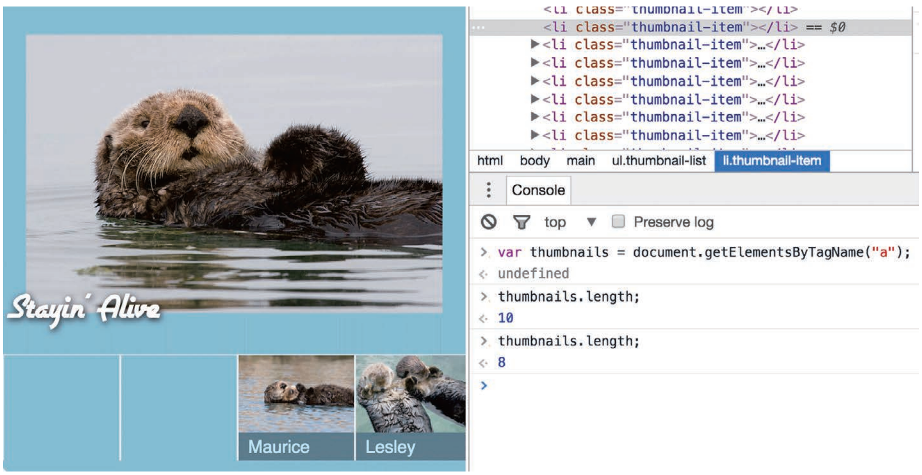


图6-30　删除元素后长度值发生了改变

将`NodeList`和`HTMLCollection`对象转换成数组不仅可以通过数组的方法更方便地操作它们，而且还可以保证即便修改DOM，数组中的值也不会发生变化。

6.17　延展阅读：JavaScript 类型

本章通过创建变量在函数内引用了一些数据。之前曾经说过字符串、数值和布尔值是五种**基本数据类型**中的三种，而另外两种则是空类型（`null`）和未定义类型（`undefined`）。

表6-2总结了这五种基本类型的性质。

表6-2　JavaScript中的五种基本数据类型

类　型	例　子	描　述
字符串	`"And you get $100! And you get $100! And...! "`	由一对引号括起来的字母、数字以及符号
数值	`42`、`3.14159`、`-1`	所有的整数及小数
布尔值	`true`、`false`	关键字为`true`和`false`，分别代表逻辑真和逻辑假
空	`null`	该值表示非法值
未定义	`undefined`	表示具有该值的对象并未被赋值

JavaScript中的其他数据类型均被认为是**复合类型**或**复杂类型**，其中包括数组和对象，它们内部都可以包含其他类型。比如编写一个函数，使其产生一个缩略图对象的数组，数组也具有一些属性（比如`length`）和方法（比如`forEach`）。

本书还将继续介绍基本数据类型和复合数据类型。

第 7 章 使用CSS营造视觉效果

在上一章中，Ottergram已经可以响应用户单击缩略图的事件，从而改变大图了。本章将继续在Ottergram上增加不同的视觉效果。

第一个效果是简单的布局变化：隐藏大图，并使缩略图与页面同宽。当用户单击一张缩略图时，大图重新出现，缩略图变回原来的尺寸。

其他两个效果则是使用CSS为缩略图和大图创建动画效果（如图7-1所示）。

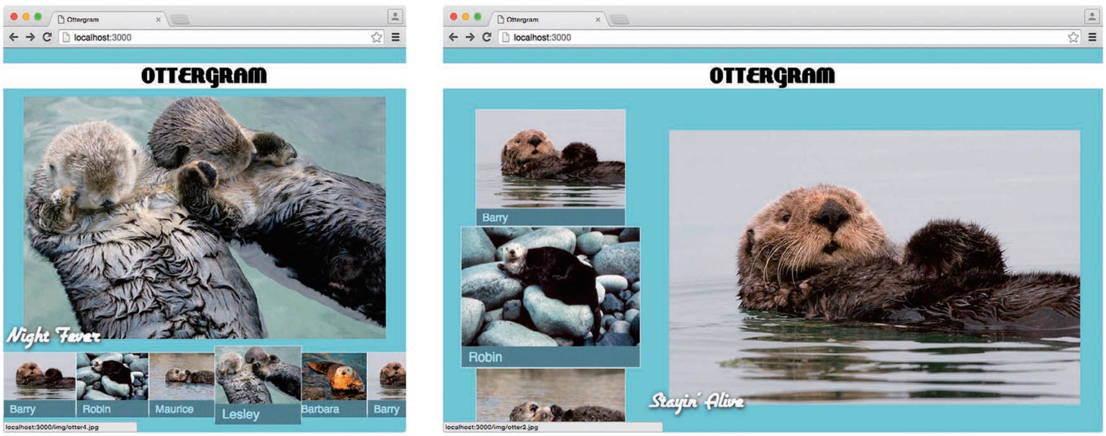


图7-1　Ottergram中的过渡效果

7.1　隐藏及显示大图

Ottergram的用户可能希望在滚动缩略图的时候，不在页面上显示大图（如图7-2所示）。

为了达到这样的效果，需要根据网站中的某个条件是否成立，为`.thumbnail-list`和`.detial-image-container`应用样式。一种方法是创建新的类选择器，比如`.thumbnail-list-no-detail`和`.hidden-detail-image-container`，并且用JavaScript把这两个类添加到目标元素上去。这个方法的问题是效率不高。

引起隐藏大图的事件也要**同时**引起缩略图列表重新调整自己的位置。这里只有一个事件，但

分别向<ul>和<div>元素中添加类无法体现这一点。

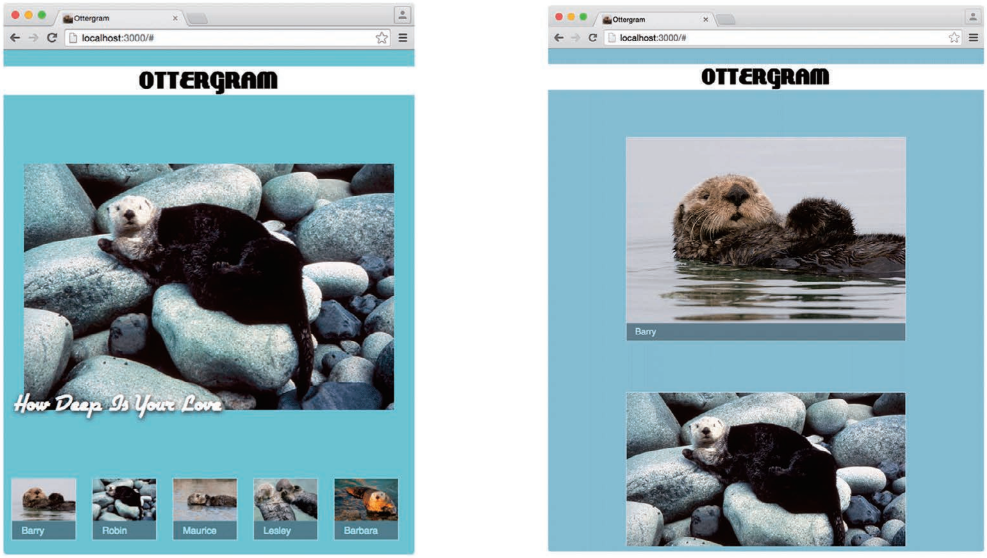


图7-2　大图的显示与隐藏

更好的方法是只使用JavaScript添加**一个**类选择器，并使其影响整个布局。然后，就可以通过这个新选择器定位作为后代的`.thumbnail-list`和`.detail-image-container`。

为此，要动态地向<body>元素添加类名，以隐藏大图并放大缩略图，然后再动态地删除该类名以返回原来的状态（如图7-3所示）。

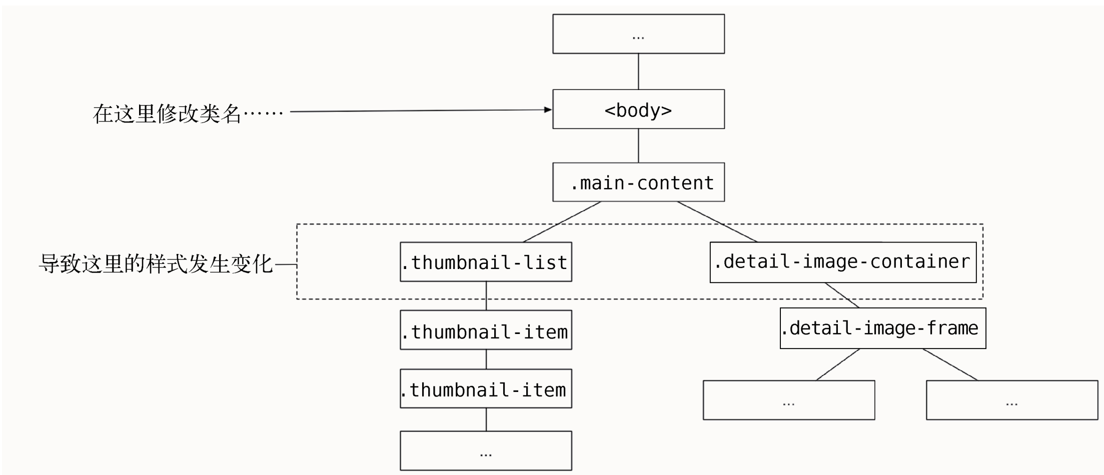


图7-3　当祖先的类发生变化时派生改变样式

这种技术与使用媒体查询有两点类似之处。

第一，都涉及在祖先符合特定条件的情况下激活某些样式。在媒体查询中，祖先即视口，而条件可能是最小宽度；在这里，祖先由任何你选择的目标元素共享，而条件为祖先有某个特殊的类名。

第二，条件样式必须放在受影响元素的其他声明之后，因为条件样式需要在它们生效时覆盖之前的声明。

你将通过3步完成。

(1) 在CSS中定义想要达到的效果的样式，同时在开发者工具中测试这些样式。

(2) 编写JavaScript函数来添加和删除`<body>`元素中的类名。

(3) 添加事件监听器来触发写好的JavaScript函数。

7.1.1　创建隐藏大图的样式

为了隐藏`.detail-image-container`，可以通过添加一条声明来将元素设为`display: none`。`display: none`可告诉浏览器该元素不应被渲染。

将被动态加入`<body>`的类名为`hidden-detail`。因此，仅当`.detail-image-container`为`.hidden-detail`的后代时才需要应用`display: none`。

在styles.css中加入隐藏大图的样式：

```
...
.detail-image-title {
  ...
}

.hidden-detail .detail-image-container {
  display: none;
}

@media all and (min-width: 768px) {
  ...
}
```

现在来想想`.thumbnail-list`应该变成什么样子。基于当前的样式，它会在宽屏幕上靠左排成一列，在窄屏幕上在顶部排为一行。而当大图被隐藏时，无论屏幕是大是小，还是让列居中更好些。

当`.thumbnail-list`和`.thumbnail-item`是`.hidden-detail`的后代时，在style.css中为它们添加如下样式：

```
...
.hidden-detail .detail-image-container {
  display: none;
}

.hidden-detail .thumbnail-list {
  flex-direction: column;
```

7

```
  align-items: center;
}

.hidden-detail .thumbnail-item {
  max-width: 80%;
}

@media all and (min-width: 768px) {
  ...
}
```

现在，thumbnail-list在.detail-image-container被隐藏时将会显示成一列。

同时，在大图被隐藏时，我们还给.thumbnail-item元素加入了max-width: 80%来设置宽度，这条声明会覆盖其他地方对于.thumbnail-item关于max-width的设置，这样一来它们便会成为页面的焦点。

当.detail-image-container、.thumbnail-list和.thumbnail-item元素成为hidden-detail类的后代时，这些新添加的样式将会被激活。

注意，这些要添加在媒体查询**之前**。如前所述，CSS的顺序很重要，文件中出现位置靠后的样式会覆盖之前的样式。通常对于同一个选择器而言，浏览器会选择它看到的最近的样式。但这里新样式使用的选择器比在媒体查询中出现的选择器更明确，而“明确”优于“最近”。

总之，最好让媒体查询出现在文件末尾，因为媒体查询总会重用已存在样式的选择器，所以把它们放在末尾可以确保媒体查询能覆盖已有样式。同时，这也更方便查找媒体查询，因为它们总出现在文件末尾。

保存文件，但在开始写应用现有样式的JavaScript之前，最好先测试一下当前的样式是否正确。启动browser-sync（使用browser-sync start --server --browser "Google Chrome" --files "*.html, stylesheets/*.css, scripts/*.js"），同时打开开发者工具。在Elements面板中，按Control键并右击<body>元素，在弹出菜单中选择Add Attribute（如图7-4所示）。

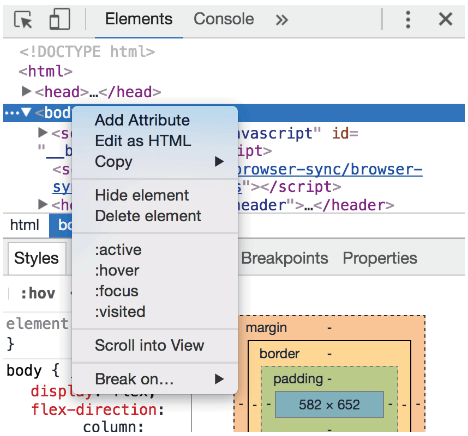


图7-4　选择Add Attribute菜单项

开发者工具提供了可以编辑<body>标签的地方，输入class="hidden-detail"并按下回车（如图7-5所示）。

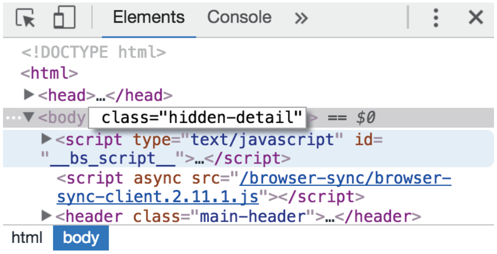


图7-5 添加hidden-detail类属性

在开发者工具中为<body>添加hidden-detail类后，大图消失，缩略图的尺寸变得大多了——一切正如所愿（如图7-6所示）。

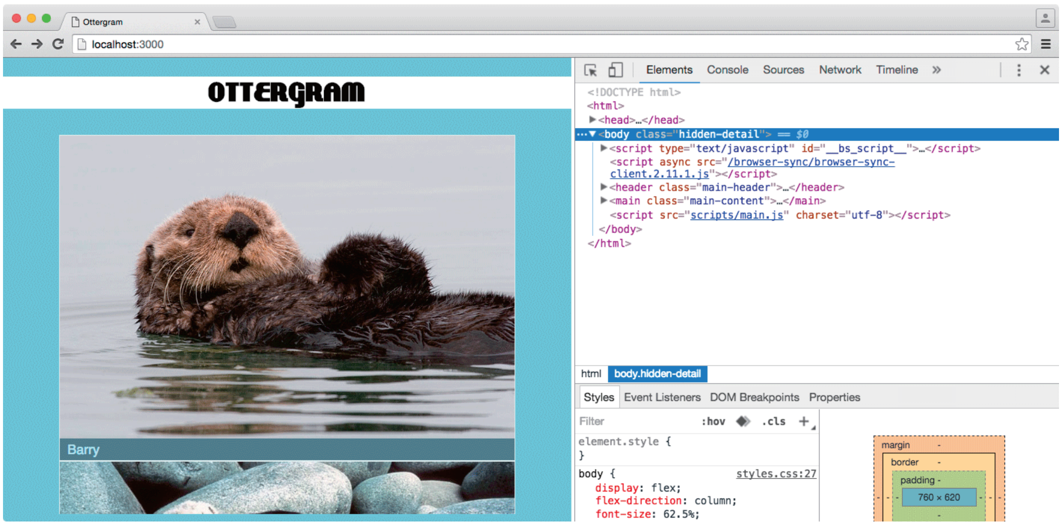


图7-6 应用hidden-detail类后的布局变化

7.1.2 用JavaScript隐藏大图

接下来编写JavaScript，切换<body>元素中的.hidden-detail类。

在main.js中添加一个名为HIDDEN_DETAIL_CLASS的变量。

```
var DETAIL_IMAGE_SELECTOR = '[data-image-role="target"]';
var DETAIL_TITLE_SELECTOR = '[data-image-role="title"]';
var THUMBNAIL_LINK_SELECTOR = '[data-image-role="trigger"]';
var HIDDEN_DETAIL_CLASS = 'hidden-detail';
...
```

现在，在main.js中编写一个名为hideDetails的函数，其作用是将一个类名加入到<body>元素中。使用DOM的classList.add方法来操作类名。

```
...
function getThumbnailsArray() {
  ...
}

function hideDetails() {
  'use strict';
  document.body.classList.add(HIDDEN_DETAIL_CLASS);
}

function initializeEvents() {
  ...
}
...
```

这里通过document.body属性来访问<body>元素，这个DOM元素与<body>标签对应。跟所有DOM元素一样，它也提供了一个简单的方法来操作类名。

这里也是通过document.body调用add方法向<body>添加hidden-detail类。

7.1.3　监听键盘事件

现在需要一个方法来触发大图的隐藏。和先前一样，需要使用一个事件监听器，但是这次的事件监听器将监听按键而非鼠标点击。

“按键”通常是指一个键按下并抬起，但这个看似简单的过程实际上会触发多个事件。当键被按下时，首先会触发keydown事件；如果按下的是一个字母键（与Shift这种功能键相对的键盘键），则还会触发keypress事件。当键被释放时，会触发keyup事件。

对Ottergram不必区分上述事件，这里将使用keyup。

在main.js中加入一个名为addKeyPressHandler的函数，它会调用document.body.addEventListener，传入一个值为'keyup'的字符串和一个匿名函数，这个匿名函数声明了一个名为event的参数。请确保在该匿名函数内event调用了perventDefault函数，并通过console.log输入event的keyCode。

```
...
function hideDetails() {
  ...
}

function addKeyPressHandler() {
  'use strict';
  document.body.addEventListener('keyup', function (event) {
    event.preventDefault();
    console.log(event.keyCode);
  });
}
```

```
function initializeEvents() {
  ...
}
...
```

所有的按键事件都有一个与触发事件的键对应的属性，称为keyCode。keyCode是一个整数，比如13是回车的keyCode，32是空格的keyCode，而38则是上箭头的keyCode。

修改main.js中的initializeEvents，让它也调用addKeyPressHandler。由此，<body>元素便可以在页面加载时监听键盘事件了。

```
...
function initializeEvents() {
  'use strict';
  var thumbnails = getThumbnailsArray();
  thumbnails.forEach(addThumbClickHandler);
  addKeyPressHandler();
}

initializeEvents();
```

保存修改并返回浏览器，确保控制台可见。点击页面使得焦点从开发者工具中移出（否则将无法触发事件监听器）。敲几下键盘上的键，控制台中会输出相应的数字（如图7-7所示）。

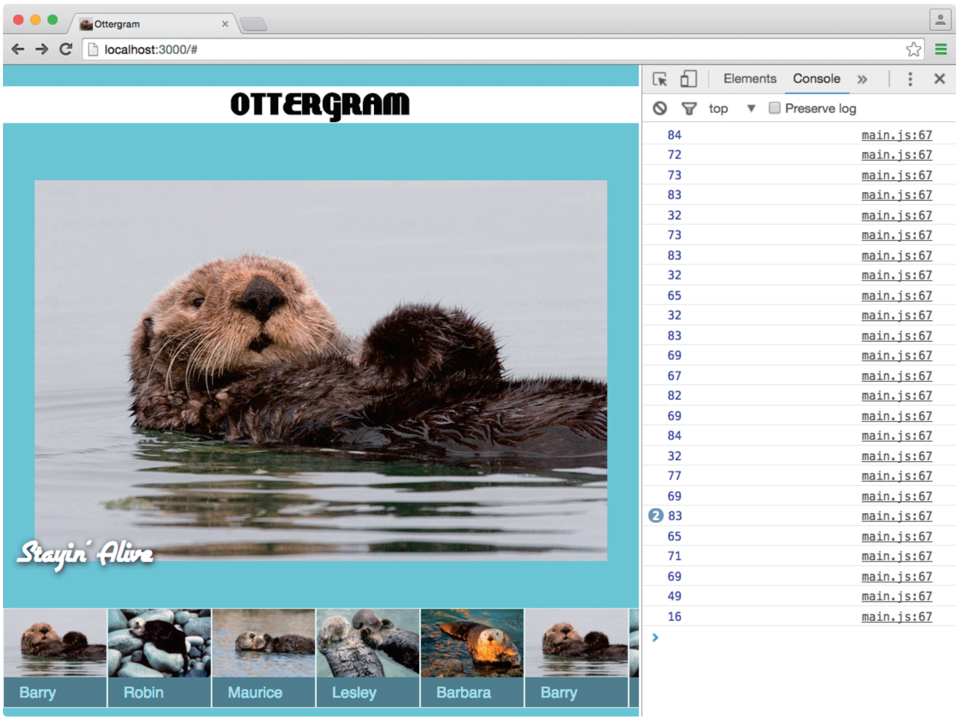


图7-7　在控制台上记录keyCode

如果只希望在按下Esc键，而非按下任意键时隐藏大图，就先敲一下Esc键——可以看到相应的`event.keyCode`是27。通过它，可以让事件监听器变得更加明确。

在main.js顶部加入一个变量来代表Esc键的值。

```
var DETAIL_IMAGE_SELECTOR = '[data-image-role="target"]';
var DETAIL_TITLE_SELECTOR = '[data-image-role="title"]';
var THUMBNAIL_LINK_SELECTOR = '[data-image-role="trigger"]';
var HIDDEN_DETAIL_CLASS = 'hidden-detail';
var ESC_KEY = 27;
...
```

修改`keyup`事件监听器，使其在`event.keyCode`的值与`ESC_KEY`的值相等时调用`hideDetails`函数。

```
...
function addKeyPressHandler() {
  'use strict';
  document.body.addEventListener('keyup', function (event) {
    event.preventDefault();
    console.log(event.keyCode);
    if (event.keyCode === ESC_KEY) {
      hideDetails();
    }
  });
}
...
```

使用**严格相等运算符**（===）来比较`event.keyCode`和`ESC_KEY`两者的值。当它们相等时，调用`hideDetails`函数。

也可以使用**松散相等运算符**（==）来做比较运算，不过通常使用严格相等运算符是更好的选择。这两个运算符的主要区别在于后者会自动进行类型转换，而前者不会。在严格相等运算符下，如果运算符两边的类型不相同，比较的结果将会是`false`。

许多前端开发者将这种自动类型转换称为**强制类型转换**，通常发生在比较（使用比较运算符）、相加（数值）或连接（字符串）两个值时。

正因为有了强制类型转换，将字符串`"27"`与数字`42`相加是没有语法错误的，尽管结果可能不似我们所想（如图7-8所示）。

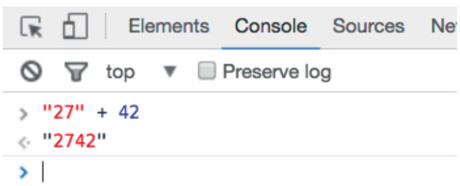


图7-8　JavaScript会自动进行类型转换

这一点在处理由用户提供的数据时极为重要，第10章将会用到与此相关的知识。

保存main.js，并在浏览器中测试新添加的功能（如图7-9所示）。

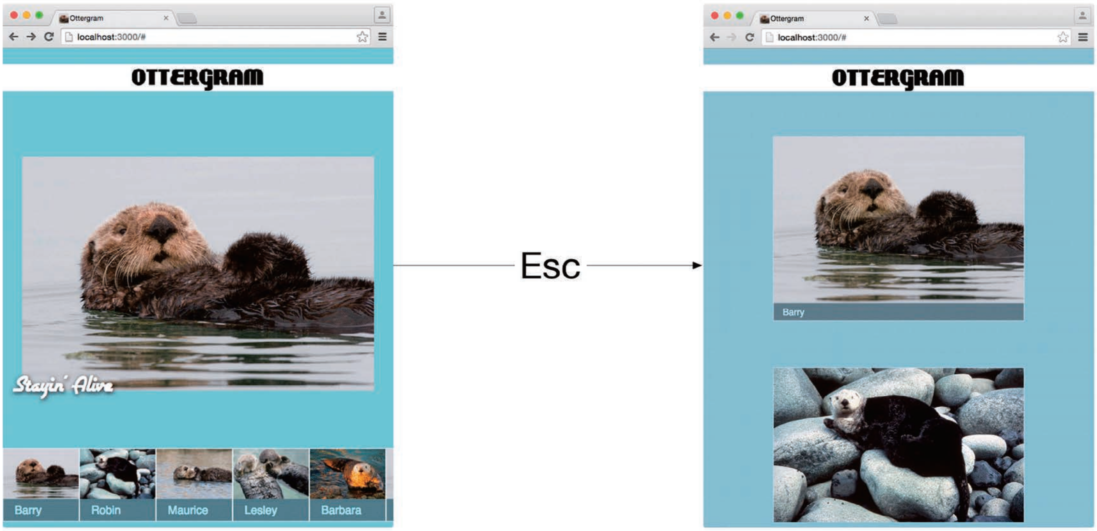


图7-9　哇！按Esc键隐藏了大图和大图标题

7.1.4　重新显示大图

有个很小但却很重要的功能需要添加：再次显示大图。这个结果由点击一张缩略图触发。

使用classList.add可以在<body>元素中添加类名；反过来，当一张缩略图被点击时，可以使用classList.remove来删除添加进去的类名。在main.js中添加一个名为showDetails的新函数。

```
...
function hideDetails() {
  ...
}

function showDetails() {
  'use strict';
  document.body.classList.remove(HIDDEN_DETAIL_CLASS);
}

function addKeyPressHandler() {
  ...
}
...
```

不需要添加新的事件监听器，只要在addThumbClickHandler函数中调用showDetails就可以了。

```
...
function addThumbClickHandler(thumb) {
  'use strict';
  thumb.addEventListener('click', function (event) {
    event.preventDefault();
    setDetailsFromThumb(thumb);
    showDetails();
  });
}
...
```

保存main.js并切换到浏览器，测试一下隐藏大图这项新功能。然后点击一张缩略图令其重新显示（如图7-10所示），水獭们看起来很喜欢这一新功能，不是吗？



图7-10　按Esc键隐藏大图，单击显示大图

使用媒体查询后，Ottergram可以根据视口尺寸动态地调整布局，并且响应用户输入。

目前布局的变化是突然发生的。而在下一节中，将使用CSS渐变来进行平滑的转变。

7.2　使用 CSS 过渡改变状态

CSS过渡可以创造从一种视觉状态变化到另一种视觉状态的效果，这正可以使Ottergram页面的隐藏/显示效果更加平滑。

创建CSS过渡时需要告诉浏览器："我希望这个元素的样式变为这些新的属性。我一告诉你，你就给我变。"

一个常见的例子就是很多网站上的弹出式菜单，比如在小屏幕上访问bignerdranch.com。在一个视口较窄的浏览器上点击菜单按钮，让导航菜单从顶端弹出——但它并不是一次性全弹出，而是从页面顶部滑下来。从初始状态（隐藏）到终止状态（可见）之间有一段明显的动画效果（如图7-11所示）。再次点击菜单图标，导航菜单又会滑上去并再次隐藏。

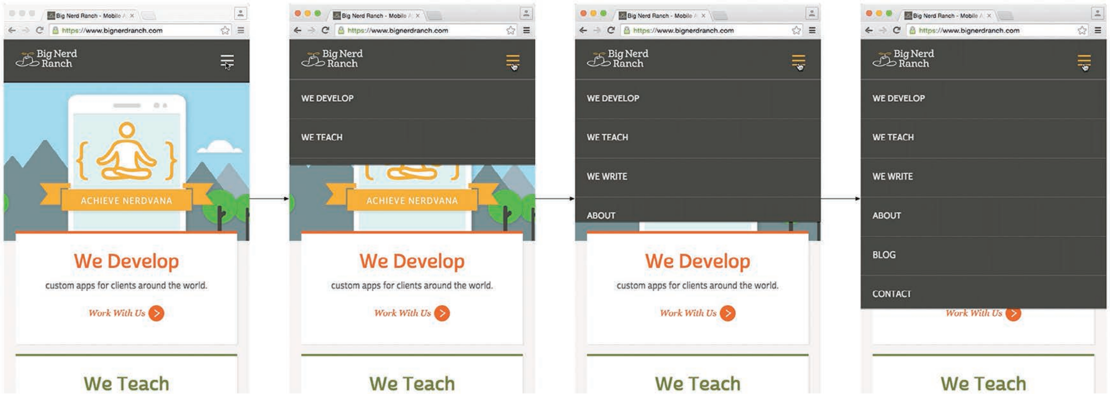


图7-11　bignerdranch.com网站上的弹出式导航

在创建显示和隐藏大图的过渡效果之前，需要先为缩略图创建一个简单的过渡。

一般来说，可以通过下面3步来创建过渡效果。

(1) 确定终止状态的样子。向目标元素添加终止状态的CSS声明是一个不错的办法，这样可以在浏览器中看见这些效果，确保它们的显示效果与设想一致。

(2) 将声明从目标元素已有的代码块移至新的CSS代码块，以便在新的块上使用新的选择器。

(3) 向目标元素添加一个过渡声明，过渡属性会告诉浏览器从当前CSS属性值到终止CSS属性值间要有一段视觉动画，并且该过渡效果应持续一段时间。

7.2.1　变形

第一个过渡效果是当光标在缩略图上悬停时其尺寸会变大（如图7-12所示）。这里并不去直接修改`width`或`height`样式，而是使用`transform`（变形）属性——它可以改变某一元素的形状、尺寸、角度以及位置，同时不影响其周围的元素。

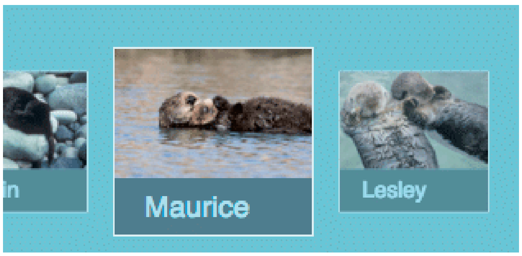


图7-12　缩放效果下的缩略图

需要进行过渡处理的目标元素是`.thumbnail-item`。首先，将`transform`声明直接添加到`.thumbnail-item`元素。

通过测试确定已实现预期效果后，应当将变形相关代码移动至新的`.thumbnail-item:`

hover声明块中。最后，再将transition声明添加到.thumbnail-item。

在styles.css中，先在.thumbnail-item中加入transform声明：

```
...
.thumbnail-item {
  display: inline-block;
  min-width: 120px;
  max-width: 120px;
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);

  transform: scale(2.2);
}
...
```

transform: scale(2.2)告诉浏览器该元素需要显示为其原尺寸的220%。在transform中还可以用到很多其他的值，包括最新的3D效果，在MDN上（developer.mozilla.org/en-US/docs/Web/CSS/transform）可以找到很好的说明。

保存并在浏览器上查看更改后的效果（如图7-13所示）

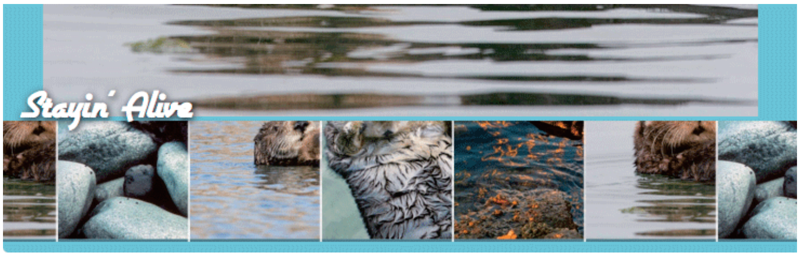


图7-13　自动变大的水獭缩略图

可以看到缩略图比之前大了，事实上它们的尺寸有点过大了，更改一下相关的值使它们小一些。

```
...
.thumbnail-item {
  display: inline-block;
  min-width: 120px;
  max-width: 120px;
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);

  transform: scale(2.2);
  transform: scale(1.2);
}
...
```

这次保存后，应该可以看到水獭缩略图的尺寸仅比它们的原尺寸大了一点点（如图7-14所示）。现在的缩略图尺寸看起来不错，可以继续进行下一步了。



图7-14 大小适中的水獭缩略图

7.2.2 添加CSS过渡效果

接下来将终止状态样式移到一个新的样式声明中，同时给`.thumbnail-item`元素添加过渡效果。

当用户的鼠标光标在一张缩略图上悬停时，该缩略图的尺寸应该被放大为原尺寸的120%。在styles.css中添加一个修饰符`:hover`的声明块来说明此样式仅应用于鼠标悬停时。

```
...
.thumbnail-item {
  display: inline-block;
  min-width: 120px;
  max-width: 120px;
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);

  transform: scale(1.2);
}

.thumbnail-item:hover {
  transform: scale(1.2);
}
...
```

这个修饰符的名称叫**伪类**，伪类`:hover`会在用户的鼠标悬停在某一元素上时与其匹配。除此之外，还有大量的伪类关键字用以描述一个元素所处的不同状态，本书后面关于表单的部分将会涉及这一部分内容，同时在MDN上可以了解到更多的相关内容。

接着，在styles.css中向`.thumbnail-item`加入`transition`声明以添加过渡效果，这里需要指定动画效果的属性以及时间长度。

```
...
.thumbnail-item {
  display: inline-block;
  min-width: 120px;
  max-width: 120px;
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);
```

```
    transition: transform 133ms;
  }

  .thumbnail-item:hover {
    transform: scale(1.2);
  }
  ...
```

以上代码为transition设置了transform属性，告诉浏览器这段样式变化要以动画形式来表现，并且动画的样式是transform。另外，代码中也指定了过渡动画应该持续133毫秒。

保存并测试一下这个新的过渡效果，应该可以看到当鼠标悬停在每张缩略图上时该缩略图会放大，而当鼠标移开时则相反，缩略图又缩小到原来的尺寸（如图7-15所示）。

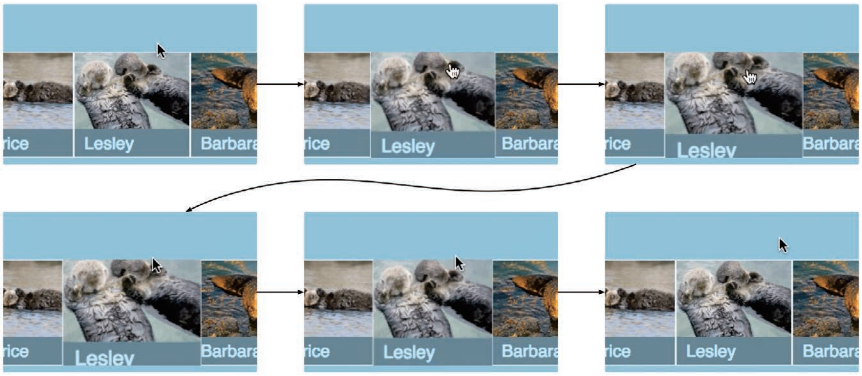


图7-15　悬停时表现出的过渡效果，移开时过渡效果相反

开发者工具提供了一个测试伪类状态的便捷方法。切换到Elements面板，展开标签直到可以看到<li>标签。单击该标签使其高亮，这时可以看到左边显示有省略号。单击该省略号，这时可以在上下文菜单中看到伪类，从中选择:hover（如图7-16所示）。

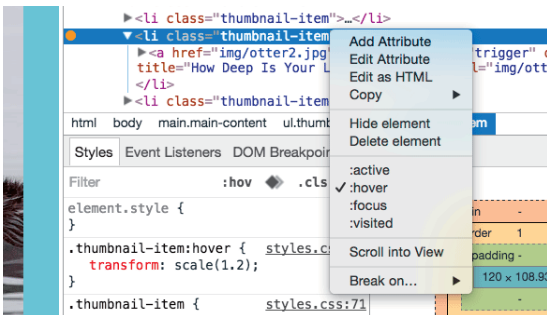


图7-16　在Elements面板中触发一个伪类

此时Elements面板中的<li>标签旁边出现了一个橙色的圆圈，表明一个伪类通过开发者工具被激活了。这种情况下，即使鼠标移上去又移开，该标签对应的缩略图仍将保持:hover状态。

单击橙色圆圈再次打开快捷菜单，禁用`:hover`状态。

过渡效果看起来很赞，不过仍然有个小小的瑕疵。目前的悬停效果会造成缩略图的一部分被截掉，这是因为应用在`.thumbnail-item`上的`transform`效果并未使得其父元素也同步调整自己的尺寸。解决方法是额外为`.thumbnail-list`添加内边距——在styles.css中更改`.thumbnail-list`的垂直内边距。

```
...
.thumbnail-list {
  flex: 0 1 auto;
  order: 2;
  display: flex;
  justify-content: space-between;
  list-style: none;
  padding: 0;
  padding: 20px 0;

  white-space: nowrap;
  overflow-x: auto;
}
...
```

上面的代码使用了内边距的简写写法，其中前面的值（`20px`）指定了上内边距和下内边距的值，而后面的值则指定了左内边距和右内边距的值。在`@media`查询中也要作出相似的修改，但同时要把左右内边距分别设为`35px`。


```
...
@media all and (min-width: 768px) {
  .main-content {
  ...
  }

  .thumbnail-list {
    flex-direction: column;
    order: 0;
    margin-left: 20px;

    padding: 0 35px;
  }
  ...
```

保存更改并在浏览器中查看新添加的效果，可以看到这次的效果比原来的好多了（如图7-17所示）。

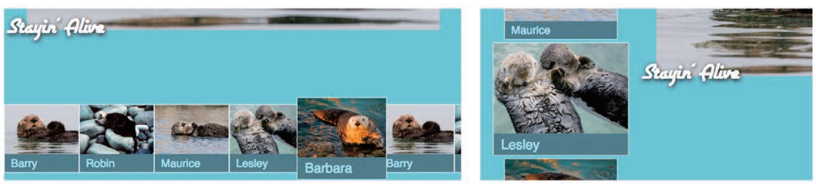


图7-17　在横向和纵向上为悬停效果留出空间

7.2.3　使用定时函数

现在的悬停效果超棒！不过仍然缺少一些让它看起来与众不同的**突出视觉效果**。通过CSS过渡效果，不仅可以规定过渡动画持续的时间，还能使动画效果在途中表现出不同的速度。

过渡效果有若干可选的定时函数，默认使用的是**线性**定时函数，即让过渡动画匀速完成。其他的定时函数则更加有意思，可以使过渡动画展现出加速或减速的效果。

在styles.css中更改先前的过渡效果——改为使用`ease-in-out`定时函数。这个函数可让过渡动画的速度在开始和结束时较慢而中间较快。

```
...
.thumbnail-item {
  display: inline-block;
  min-width: 120px;
  max-width: 120px;
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);

  transition: transform 133ms ease-in-out;
}
...
```

保存修改并拿一张缩略图测试一下。可以看到效果虽然不甚明显，不过的确有变化。

除此之外还有很多可供使用的定时函数，这部分可以参阅MDN（developer.mozilla.org/en-US/docs/Web/CSS/transition-timing-function）。

现在的过渡样式动画在**开始**和**结束**时的速度是相同的。但这也可以更改：根据过渡的方向来设定不同的值。当在初始状态和终止状态的声明中指定`transition`属性后，浏览器便会在表现过渡动画效果时按时间顺序使用声明中已经给定的值。

下面通过一个简单的例子来解释一下。首先在styles.css中向`.thumbnail-item:hover`添加`transition`声明（可以先在浏览器中更改，方便以后删除）。

```
...
.thumbnail-item:hover {
  transform: scale(1.2);
  transition: transform 1000ms ease-in;
}
...
```

保存修改并在浏览器中将光标悬停在其中一个缩略图上，可以看出放大效果变慢了，需要整整1秒钟，这是因为该效果使用了在`.thumbnail-item: hover`中声明的值。将光标从缩略图上移开，可以看到此时的动画效果仅为133毫秒，即在`.thumbnail-item`中声明的值。

在继续之前，先删除刚才对`.thumbnail-item: hover`所做的更改。

```
...
.thumbnail-item:hover {
  transform: scale(1.2);
  transition: transform 1000ms ease-in;
}
...
```

7.2.4　基于类的过渡效果

第二个过渡效果是让`.detail-image-frame`看起来像是从很远的地方拉近的。

这次将通过使用JavaScript添加/删除类名（而不是使用伪类选择器）来触发过渡效果。为什么呢？因为没有与单击事件对应的伪类，使用JavaScript可以在UI变化和触发时实现更好的控制。

另外，还可以为过渡效果的开始阶段和结束阶段设定不同的持续时间。最后的效果是：单击一张缩略图，相应的水獭图片便会作为大图展示出来，从大图区域中心一个很小的点开始，逐渐放大至其原始尺寸（如图7-18所示）。

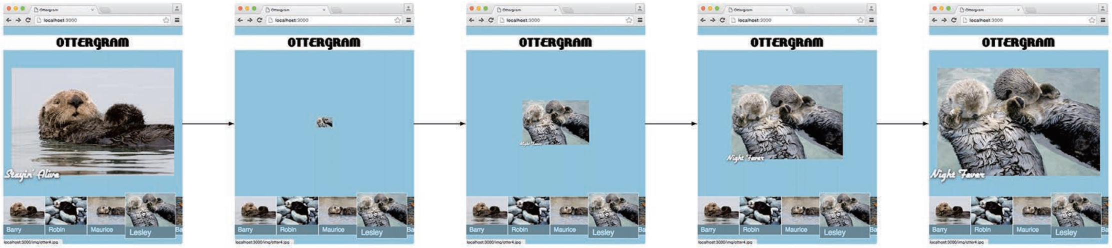


图7-18　单击一张缩略图，使其从小图放大到原始大小

首先，在styles.css中加入一个新的类样式声明`is-tiny`。

```
...
.detail-image-frame {
  ...
}

.is-tiny {
  transform: scale(0.001);
  transition: transform 0ms;
}

.detail-image {
  ...
```

上面的代码为`.is-tiny`添加了两个样式。第一个样式将元素缩小至相对于其原尺寸来说很小的一个区块，而第二个样式指定了该`transform`属性过渡效果的时长为0毫秒，也就是立即变得非常小。换句话说，在`.is-tiny`类的样式中，大图实际上是没有过渡动画效果的，因为该动画持续时间为0毫秒，没必要为其指定定时函数。

接下来，添加一个持续时间为333毫秒的`transition`声明，这个值将用于从`.is-tiny`类的样式过渡到下一个样式的动画效果，即在三分之一秒内将大图放大到正常尺寸。将`transition`的声明添加到styles.css的`.detail-image-frame`中。

```
...
.detail-image-frame {
  position: relative;
  text-align: center;
```

```
  transition: transform 333ms;
}
...
```

先保存对styles.css的修改再继续。

7.2.5　通过JavaScript触发过渡效果

过渡效果的样式已经写好了，下一步需要做的是使用JavaScript来触发它们。在index.html的`.detail-image-frame`中添加一个数据属性，使其与JavaScript挂钩。

```
...
      <div class="detail-image-container">
        <div class="detail-image-frame" data-image-role="frame">
    <img class="detail-image" data-image-role="target" src="img/otter1.jpg" alt="">
    <span class="detail-image-title" data-image-role="title">Stayin' Alive</span>
        </div>
      </div>
...
```

保存index.html，然后在main.js中为`.is-tiny`类和`data-image-role="frame"`选择器添加变量。完成后，修改`showDetails`函数，使其切换类名，从而触发过渡动画。

首先添加一个名为`DETAIL_FRAME_SELECTOR`的变量，其值为选择器字符串`'[data-image-role="frame"]'`。再添加一个名为`TINY_EFFECT_CLASS`的变量，其值为`is-tiny`。

```
var DETAIL_IMAGE_SELECTOR = '[data-image-role="target"]';
var DETAIL_TITLE_SELECTOR = '[data-image-role="title"]';
var DETAIL_FRAME_SELECTOR = '[data-image-role="frame"]';
var THUMBNAIL_LINK_SELECTOR = '[data-image-role="trigger"]';
var HIDDEN_DETAIL_CLASS = 'hidden-detail';
var TINY_EFFECT_CLASS = 'is-tiny';
var ESC_KEY = 27;
...
```

这两个变量的顺序无关紧要（对浏览器而言没有差别），不过按照相同规则排序可读性更好些。在main.js中，选择器变量后面都是类变量，最后是Esc键的键值。

修改main.js中`showDetails`函数，取得元素`[data-image-role="frame"]`的引用。为了触发`transition`类，需要先添加`TINY_EFFECT_CLASS`然后再删除它。

```
...
function showDetails() {
  'use strict';
  var frame = document.querySelector(DETAIL_FRAME_SELECTOR);
  document.body.classList.remove(HIDDEN_DETAIL_CLASS);
  frame.classList.add(TINY_EFFECT_CLASS);
  frame.classList.remove(TINY_EFFECT_CLASS);
}
...
```

如果将上述改动保存并在浏览器中测试的话，将会看到过渡效果并没有出现，为什么？原因

是TINY_EFFECT_CLASS在添加后立刻又被删掉了，最终结果就是没有可被渲染的类变化，这是浏览器端的一个优化措施。

因此需要在删除TINY_EFFECT_CLASS之前加一个短延迟，但JavaScript并没有像其他语言一样提供一个内建的延迟或睡眠函数。嗯，看来得想个变通策略了。

现在，使用setTimeout方法，其参数是一个函数名和一段延迟时间（以毫秒计）。延迟时间过后，该函数将进入相应队列等待浏览器执行。

在main.js中调用frame.classList.add函数代码的后面添加一个对setTimeout的调用，向其传入两个参数：包含接下来要执行步骤的函数，和调用该函数前等待的时间。本例中接下来要执行的仅有一步，即删除TINY_EFFECT_CLASS。

```
...
function showDetails() {
  'use strict';
  var frame = document.querySelector(DETAIL_FRAME_SELECTOR);
  document.body.classList.remove(HIDDEN_DETAIL_CLASS);
  frame.classList.add(TINY_EFFECT_CLASS);
  setTimeout(function () {
    frame.classList.remove(TINY_EFFECT_CLASS);
  }, 50);
}
...
```

仔细阅读上面的代码。首先，向frame元素添加.is-tiny类，其效果便是transform: scale(0.001)。

浏览器等待50毫秒后向自己的执行队列中加入一个匿名函数。执行showDetails函数，50毫秒后匿名函数入队等待执行（该函数要等待它前面的函数释放所占用的CPU资源）。

当该匿名函数执行时，它会从frame的类列表中删除TINY_EFFECT_CLASS。这会触发长度为333毫秒的transform动画效果，最终frame的尺寸会扩大至其原始大小。

保存本次修改并测试下效果。单击缩略图，可以看到这些滑稽的水獭图片逐渐放大直至可供欣赏。

7.3 自定义定时函数

下面为Ottergram锦上添花：为过渡效果自定义一个定时函数来代替有限的内置定时函数。

定时函数可以通过曲线图来说明，内置定时函数的曲线图（来源网站：cubic-bezier.com）如图7-19所示。



图7-19 内置的定时函数

7

上面的5幅图像中的曲线被称为**三次贝塞尔曲线**，描述了过渡动画随时间变化的规律。它们由4个控制点定义。通过确定4个控制点来确定一条曲线，就可以自定义过渡效果了。下面在styles.css中为`.detail-image-frame`添加带`cubic-bezier`函数的`transition`声明。

```
...
.detail-image-frame {
  position: relative;
  text-align: center;

  transition: transform 333ms cubic-bezier(1,.06,.28,1);
}
...
```

保存修改并在浏览器中单击几张缩略图，查看过渡效果是否有了变化。

多亏了开发者Lea Verou和她的网站（cubic-bezier.com）提供的实用工具，自定义定时函数变得轻松愉快了许多（如图7-20所示）。

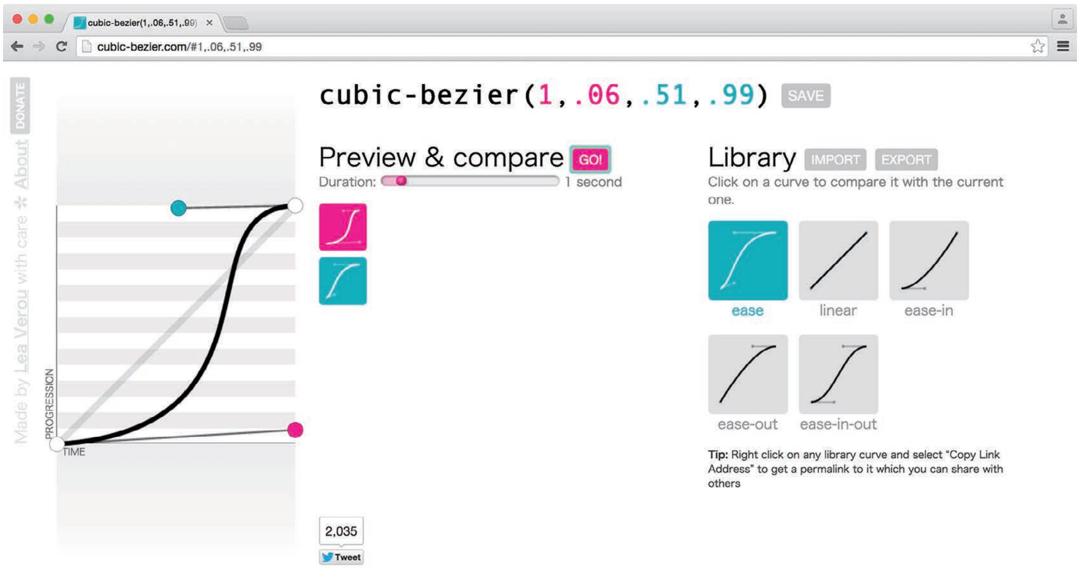


图7-20　在cubic-bezier.com上自定义定时函数

左侧是一条带有红色和蓝色可拖动控制点的曲线，该曲线表示在当前阶段进行转变的程度。单击并拖拽控制点来修改曲线。当图像改变时，页面顶端表示控制点值的10进制数字也会改变。

右侧是内置的定时函数：`ease`、`linear`、`ease-in`、`ease-out`和`ease-in-out`。单击其中的一个，然后单击Preview&compare旁边的GO!按钮。接着就会展示自定义定时函数所表现出的动画效果，并且与内置选项进行对比。

创建一个自定义定时函数，感到满意后将三次贝塞尔曲线控制点的值从网站上复制下来，粘贴到styles.css中的相应位置。

```
...
.detail-image-frame {
  position: relative;
  text-align: center;

  transition: transform 333ms cubic-bezier(把自定义的值粘贴到这里);
}
...
```

恭喜！Ottergram网站已功能齐备。保存相关文件并且欣赏一下最终的效果吧。Ottergram已经从一个简单、静态的网页变成了一个可交互、可响应并带有动画视觉效果的页面。

整个项目已经取得了很大的进展，相信你也从中学到了很多前端开发的基础知识。该跟水獭们说再见了，因为下一章我们又会开始一个新项目。

7.4 延展阅读：强制类型转换的规则

正如在第6章中所提到的，JavaScript最初的开发宗旨是使普通人，而不是专业的程序员可以为网页添加交互元素。而一个“普通人”是不应该担心一个值到底是数值，是对象，或者是个香蕉的。（开个玩笑，JavaScript中没有香蕉这个类型。）

达成上述目的的方法之一便是强制类型转换。通过强制类型转换，可以直接使用==操作符对两个值进行比较，或是使用+操作符对两个值进行连接，而不用去管它们的类型。在进行这类操作时，JavaScript会设法使其成功——即使结果看起来有点奇怪，比如把字符串"2"转换成数值2。

7

强制类型转换对程序员和非程序员都是个困扰。大部分程序员认为最好还是使用严格比较，即===比较好。不过强制类型转换的规则在语法中早已被严格定义过了，因此有必要了解一下。

假设现在要对两个变量进行比较：`x == y`。如果它们的类型相同，并且值相等，比较结果将会返回一个值为`true`的布尔值。唯一的例外是如果x或y的值为NaN时（语法中规定该常数表示“非数值”），其比较值为`false`。

然而，当x和y的类型不同时，事情就复杂了。下面便是JavaScript中的相关规则。

- 当比较表达式`null == undefined`和`undefined == null`时，值为`true`。
- 当比较一个字符串和一个数值时，首先将字符串转换成等价的数值形式。这意味着`"3" == 3`的值为`true`，而`"dog" == 20`的值为`false`。
- 当比较布尔值与其他类型值时，首先将布尔值转换成数值：`true`转换成`1`，`false`转换成`0`。这意味着`false == 0`的值为`true`，而`true == 1`的值也为`true`。
- 最后，将字符串或数值与对象进行比较时，首先试着将对象转换成一个基本类型的值；如果无法转换，再试着将该对象转换成一个字符串。

更多相关内容，请查阅MDN上的讨论（developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness）。

第二部分

模块、对象及表单

第 8 章

模块、对象和方法

在接下来的7章中，我们将为美食车搭建一个管理咖啡订单的应用，名字叫作CoffeeRun。需要编写的代码分三个层级：UI、内部逻辑和服务端交换逻辑。

本章将会说明如何创建CoffeeRun内部逻辑，并且通过开发工具的控制台与应用进行交互，如图8-1所示。

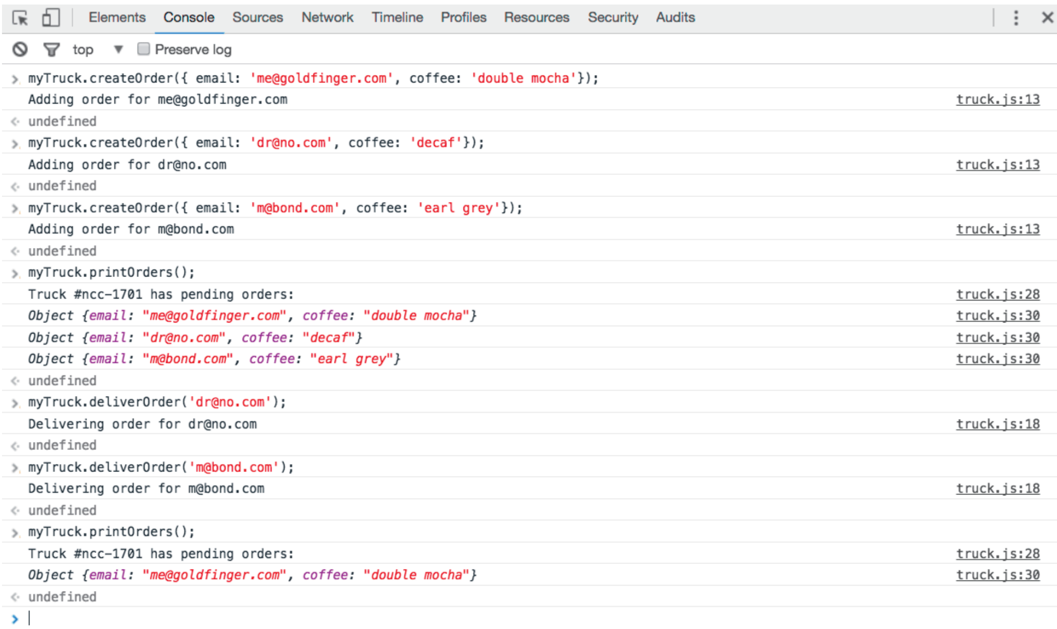


图8-1　CoffeeRun的底层逻辑

8.1　模块

CoffeeRun比Ottergram复杂了许多，因此能否合理地组织代码对调试和拓展十分重要。CoffeeRun将会以组件的方式组合起来，如图8-2所示。

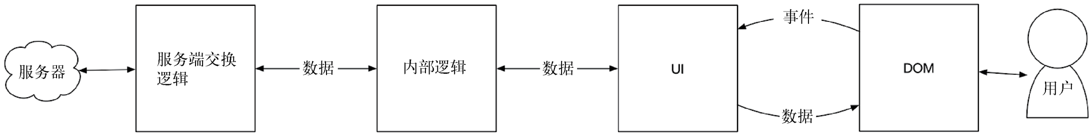


图8-2 CoffeeRun组件和交互一览图

应用中的每一部分都会专注于一个任务：内部逻辑模块负责处理数据，UI模块负责处理事件和DOM操作（和Ottergram的代码相似），服务端交换逻辑模块负责和远端服务器交互，保存或者检索数据。

JavaScript的设计初衷是编写负责用户交互的小型脚本，而非大型应用。虽然CoffeeRun并不复杂，但仅仅一个脚本文件并不足以完成它。

为了让代码彻底分离，需要创建三个不同的JavaScript文件，分别对应UI、内部逻辑和服务端交换逻辑。我们将以**模块**的形式实施代码分离。

怎样将代码组织成模块完全取决于开发者。大多数情况下都以概念切分代码，如“广告”或者“菜单”。就代码而言，模块是一组相关函数的集合，部分函数可以被外部访问，其余函数则只能被模块内部访问。

来想象一家餐厅。它的厨房里有各种供厨师使用的工具和配料，而消费者只能看菜单上的若干选项。如果将厨房比作食物制作模块的话，那么这个菜单就是消费者和厨房进行交互的接口。

同样，如果餐厅内部拥有一个吧台，那么饮料制作工具和配料也是内部的，消费者只能通过饮品单进行选择，所以也可以认为饮品单就是吧台和消费者间交互的接口。

作为消费者，我们既不能从厨房里借来大厨的菜刀或者使用吧台的搅拌机，也不能从冰箱里再拿出一份黄油或者为鸡尾酒再多加一杯配酒。我们的选择受限于厨房和吧台提供的菜单。

与之相似，CoffeeRun的每一个模块也需要一些私有函数——只公开一部分函数以便其余模块与其进行交互。

CoffeeRun将继续使用ES5，这是能实现本项目且在目前浏览器中支持度最高的JavaScript版本（下一个项目Chattrbox将会使用最新的ES6）。ES5并没有提供正式的模块化方案，但通过将相关的代码（变量和函数）放在一个函数内，可以获得类似的效果。

8.1.1 模块模式

代码是可以用函数进行组织的，但是通常采用常规函数的变体来实现这一目的。在开始新项目前，先快速浏览用于组织代码的**模块模式**，尝试一下将Ottergram里的函数包装成模块。

以下是一个基本模块的代码：

```
(function () {
  'use strict';
  // 放置要运行的代码
})();
```

如果你是第一次看见这种写法的代码，可能会感到奇怪。这种写法被称作**立即调用的函数表**

达式（Immediately Invoked Function Expression，IIFE），最好从内到外地去阅读它。

它最主要的部分是一个匿名函数：

```
function () {
  'use strict';
  // 放置要运行的代码
}
```

之前在Ottergram中使用过匿名函数，所以你应该不会对此感到陌生。

然而在本例中，匿名函数被括号所包裹：

```
(function () {
  'use strict';
  // 放置要运行的代码
})
```

这对括号十分重要，因为它们告诉浏览器："请不要将这组代码解释为函数声明。"

浏览器看到这个括号就会知道："啊，好的。这是一个匿名函数，我会阻止它做任何事情的"。

在大多数情况中，我们会将匿名函数当作一个参数进行传递。而在这个例子里，我们马上就调用了它。这是通过它后面的的空括号做到的：

```
(function () {
  'use strict';
  // 放置要运行的代码
})()
```

当浏览器读到空括号的时候，它意识到你希望调用括号前的任何东西："好吧，我明白了。我现在有一个匿名函数可以调用。"

你可能会认为："这是疯了吧！一点用处都没有。"事实上，你已经写过类似的代码了。回忆一下Ottergram的初始化函数`initializeEvents`——声明它之后，我们立马调用了它，并且之后再也没有调用过。以下就是当时的代码：

```
function initializeEvents() {                // 函数声明
  'use strict';
  var thumbnails = getThumbnailsArray();
  thumbnails.forEach(addThumbClickHandler);

  addKeyPressHandler();
}

initializeEvents();                          // 函数执行
```

这个函数的作用就是将一些步骤捆绑在一起，并且在页面加载的时候使其运行。下面是使用IIFE模式编写的相同的代码。（你并不需要更改Ottergram的代码，这只是为了解释概念。）

```
function initializeEvents() {
(function () {
  'use strict';
  var thumbnails = getThumbnailsArray();
  thumbnails.forEach(addThumbClickHandler);
```

```
  addKeyPressHandler();
}
})();

initializeEvents();
```

如果我们想在不创建额外的全局变量或者函数的前提下只运行某些代码一次，IIFE模式是十分有用的写法。要理解它为什么这么重要，需要回顾一下在Ottergram中编写的变量和函数。

我们在Ottergram中创建了许多有用的函数，并且在需要的时候调用了它们。这些函数都有像getThumbnailsArray或者addKeyPressHandler之类的名称。很幸运，这些名称是唯一的。如果我们尝试在两个不同函数上使用相同的名称，第一个函数就会被第二个函数取代。

当我们定义函数和变量时，它们默认被添加到**全局命名空间**中。全局命名空间是浏览器为JavaScript程序存储所有函数和变量的地方，同时也是存储所有内建函数和变量的地方。一般会使用**命名空间**来组织代码，就像会把文件放在文件夹里一样。

在CoffeeRun中可能会有很多名称相同的函数，如add或者addClickHandler。如果把它们都写在全局命名空间中，它们会互相覆盖。因此，应该在函数内声明它们，这能避免它们被外界的代码访问或者覆盖。

因为我们是在模块内组织代码，所以可能会想让一部分（不是全部）函数能被外界访问。为了达成这个目的，需要利用IIFE可以传参的特性。

8.1.2 通过IIFE修改对象

IIFE不仅仅擅长运行像Ottergram项目中的initializeEvents那样的初始代码，它们还能改造作为参数传入的对象。为了解释清楚它是怎么做到的，我们的老朋友initializeEvents又要出场了。

本例中的initializeEvents为缩略图添加点击事件。

（为了简要说明，移除了对addKeyPressHandler的调用。）

```
function initializeEvents() {
  'use strict';
  var thumbnails = getThumbnailsArray();
  thumbnails.forEach(addThumbClickHandler);
}

initializeEvents();
```

在这种形式下，initializeEvents通过addThumbClickHandler修改一组缩略图的同时，也能通过实参接受数组。首先在定义函数时声明一个形参，然后在调用它时再传入一个数组，如下：

```
function initializeEvents(thumbnails) {
  'use strict';
  var thumbnails = getThumbnailsArray();
  thumbnails.forEach(addThumbClickHandler);
}
```

8

```
var thumbnails = getThumbnailsArray();
initializeEvents(thumbnails);
```

为了将这串代码编写成IIFE模式，需要删除函数名，用括号包裹函数，再添加一个空括号用于执行函数：

```
(function initializeEvents(thumbnails) {
  'use strict';
  thumbnails.forEach(addThumbClickHandler);
})();

var thumbnails = getThumbnailsArray();
initializeEvents(thumbnails);
```

不过我们仍然需要将一组缩略图作为实参传入，这时只需移动一下`getThumbnailsArray`即可。这样子，我们就能将结果传递到IIFE中：

```
(function (thumbnails) {
  'use strict';
  thumbnails.forEach(addThumbClickHandler);
})(getThumbnailsArray());

var thumbnails = getThumbnailsArray();
```

在这个版本的代码中，一个（通过调用`getThumbnailsArray`产生的）数组被传入到IIFE中。IIFE接受这个数组并标记为`thumbnails`。在IIFE函数中，数组中的每个元素都被绑定了事件监听器（如图8-3所示）。

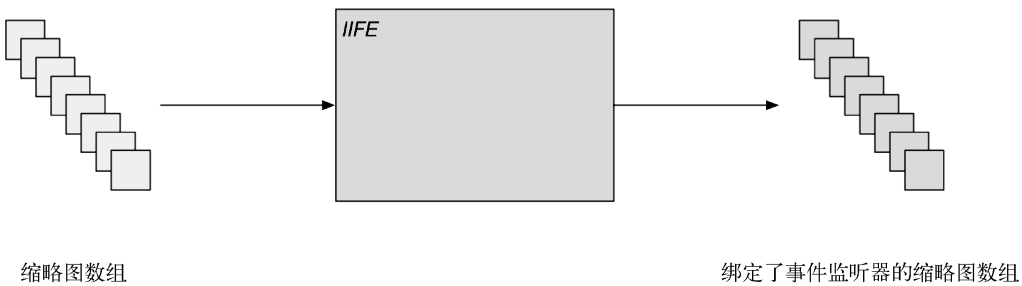


图8-3　IIFE修改其参数

我们可以把任何东西传进IIFE然后修改它。CoffeeRun中的IIFE将会接受一个`window`对象。但我们并不打算把模块代码直接放到全局命名空间上，而是要将它放在全局命名空间内的App属性上。CoffeeRun里的每个模块都有自己的文件，并且通过独立的`<script>`标签引入。图8-4展示了这个过程。

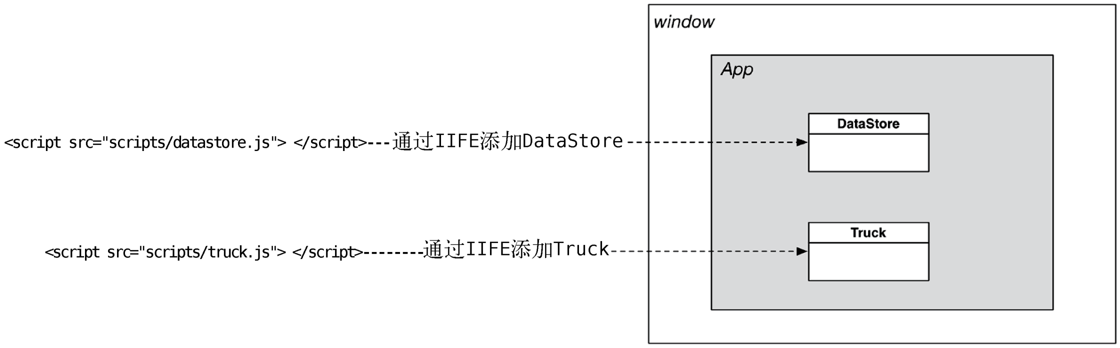


图8-4　使用<script>标签引入代码并修改window.App属性

8.2　搭建我们的 CoffeeRun 吧

理论知识已经够多了，下面开始动手干活吧。因为这是一个新项目，所以要在新目录下打开它：打开Atom编辑器，选择File → Add Project Folder，再选择front-env-dev-book目录，然后点击New Folder，将其命名为coffeerun再点击Open。

接下来，在Atom的导航栏上按下Control键，同时右击coffeerun文件夹，选择New File并将其命名为index.html。再次按下Control键，同时右击coffeerun，然后新建一个名为scripts的文件夹。

现在的front-env-dev-book文件夹中应该有ottergram和coffeerun这两个结构相似的文件夹，它们分别对应两个项目（如图8-5所示）。

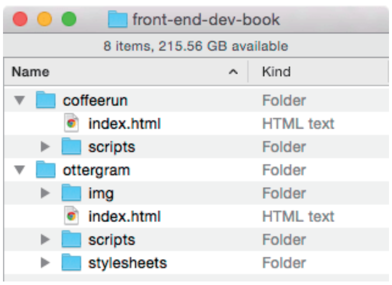


图8-5　为CoffeeRun创建文件夹和文件

如果你已经打开了终端并且正在运行browser-sync，请按下Control + C停止运行；如果没有的话，请打开一个新的终端窗口。无论使用哪种方式，将路径切换到coffeerun文件夹（如果不记得怎么做了可以翻阅第1章），然后再次启动browser-sync。提醒一下，启动命令是`browser-sync start --server -- files "stylesheets/*.css, scripts/*.js, *.html"`。

在index.html中添加一些基本代码。（记住，Atom的自动补全功能可以节省大量的工作，只要输入html即可。）

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>coffeerun</title>
  </head>
  <body>

  </body>
</html>
```

终于要编写第一个模块了！

8.3　创建数据存储模块

我们编写的第一个模块将用于将咖啡订单存储到数据库中。每个订单都会存储用户的邮箱地址。在刚开始的时候，只需要记录订单的文字描述即可，例如“四倍浓缩咖啡”（如图8-6所示）。

邮箱地址	订单
caquino@bignerdranch.com	四倍浓缩咖啡
tgandee@bignerdranch.com	黑咖啡
jreece@bignerdranch.com	咖啡果昔

图8-6　CoffeeRun数据库的初始数据

稍后还要记录每杯咖啡的杯型、口味和咖啡因浓度。客户的邮箱地址会成为每张订单的独立标记，因此每个订单都会与一个邮箱地址关联。（对不起，为了防止上瘾，每位顾客只能下一次单！）

创建一个名为scripts/datastore.js的新文件。接下来，在index.html上添加相应的`<script>`标签引入新文件。

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>coffeerun</title>
  </head>
  <body>
    <script src="scripts/datastore.js" charset="utf-8"></script>
  </body>
</html>
```

保存index.html，在scripts/datastore.js中用基本的IIFE模式编写模块。

```
(function (window) {
  'use strict';
  // 放置要运行的代码
})(window);
```

模块的架构现已准备完毕，也有对应的<script>标签，是时候将它添加到应用的命名空间上了。

8.4　在命名空间上添加一个模块

许多编程语言都有特定的语法用于建立和组织模块，然而ES5是个例外。因此，需要使用对象来构建类似的结构。

对象能将任意数据和键名关联到一起，而这正是我们组织模块的方式。具体来说，我们会用一个对象来作为CoffeeRun的命名空间，这个命名空间是各个模块注册的地方，这样它们便可以被其他应用代码所调用。

利用IIFE来建立命名空间需要3步。

(1) 如果命名空间已经存在，获取它的引用。

(2) 创建模块代码。

(3) 将模块代码绑定到命名空间上。

来看看现实中是怎么样的。将datastore.js中的IIFE部分改为下面的代码，稍后将解释为什么要这么做。

```
(function (window) {
  'use strict';
  // 放置要运行的代码
  var App = window.App || {};

  function DataStore() {
    console.log('running the DataStore function');
  }

  App.DataStore = DataStore;
  window.App = App;
})(window);
```

在IIFE中声明一个本地变量App。如果在window上存在App属性，那么就将它赋值到本地App中；否则引用一个用{}表示的新的空对象。||是**默认运算符**，被称作**逻辑或运算符**，它可以在第一个选项（window.App）尚未创建时提供一个有效值（在本例中是{}）。

我们接下来编写的每一个模块都会做相似的检验，就好比“无论谁是第一个，创建一个新对象让其余人使用”。

下一步，声明一个名为DataStore的函数。我们马上就会为这个函数添加代码。

最后，将DataStore绑定到App对象上，然后将新修改的App赋值到全局App属性上。（如果App对象不存在，就必须要新建一个空对象并绑定到全局）。

保存文件，切换到浏览器。打开开发者工具，点击Console选项卡，输入以下代码调用DataStore函数：

```
App.DataStore();
```

8

DataStore就会被执行并且在控制台输出以下信息（如图8-7所示）。

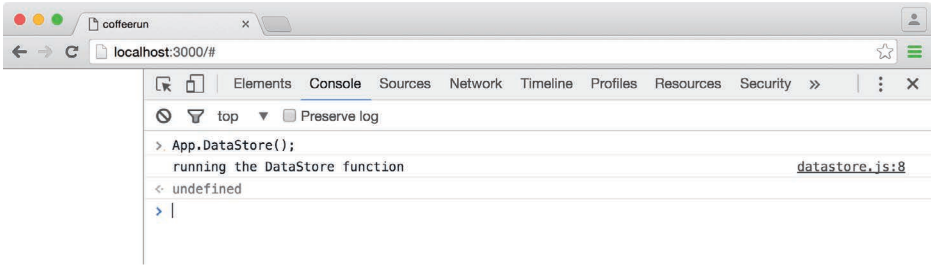


图8-7　运行App.DataStore函数

注意，不需要输入window.App.DataStore();，因为window对象就是全局命名空间，它的所有属性我们都可以直接输入，包括在控制台里。

8.5　构造函数

通过IIFE，我们可以利用函数作用域为大段代码创建命名空间。还有另外一种函数编写方式，它就像工厂一样，生产出拥有类似属性和方法的对象。其他语言可能会使用类构建相似的结构，但严格来说JavaScript并没有类的概念，不过它允许我们自定义类型。

我们已经创建了DateStore类型，现在要分两步定制它。第1步，定义用于内部存储数据的属性；第2步，提供一系列与该数据交互的方法。我们不需要给其他对象直接访问该数据的权利，因为我们会提供相应的外部接口。

在JavaScript中，工厂模式的函数被称为**构造函数**。

为datastore.js中的DataStore函数添加以下代码。

```
(function (window) {
  'use strict';
  var App = window.App || {};

  function DataStore() {
    console.log('running the DataStore function');
    this.data = {};
  }

  App.DataStore = DataStore;
  window.App = App;
})(window);
```

构造函数的任务就是创建和定制一个新的对象。在构造函数的内部，通过this关键字调用这个新对象。同时，可以通过点运算符为新对象创建一个data属性，并且分配给它一个空对象。

你可能会注意到，DataStore的首字母大写了，这是JavaScript中约定的构造函数的命名方式。虽然不是必须的，但这是一种很好的做法，因为它能告知其他开发者这是一个构造函数。

要区分构造函数和普通函数，可以观察它在被调用时有否使用了new关键字。new会告知JavaScript创建一个新对象，将this指向新对象，然后**隐式**返回该对象。这意味着即便不在构造函数中使用return语句，它也依然可以“return”一个对象。

保存文件，然后切换到控制台。为了学会如何使用构造函数，我们将会创建两个DataStore对象（又被称作**实例**），再为它们添加数据。编写代码如下：

```
var dsOne = new App.DataStore();
var dsTwo = new App.DataStore();
```

可以通过调用DataStore构造函数来创建DataStore实例。这时它们都有一个空的data属性，我们给它们添加一些数据。

```
dsOne.data['email'] = 'james@bond.com';
dsOne.data['order'] = 'black coffee';
dsTwo.data['email'] = 'moneypenny@bond.com';
dsTwo.data['order'] = 'chai tea';
```

然后查看这些数值：

```
dsOne.data;
dsTwo.data;
```

由结果可知，每个实例都保存了不同的信息（如图8-8所示）。

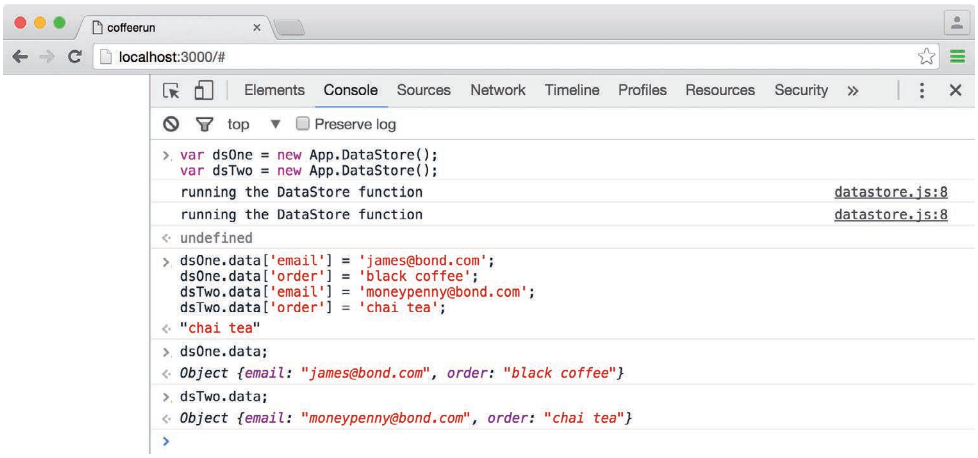


图8-8　使用DataStore构造函数创建的实例保存数据

8.5.1　构造函数的原型

使用DataStore实例可以手动存储和提取数据。但通过这种方式，DataStore只是创建了一个对象，而其余希望使用DataStore实例的模块只能直接操作DataStore的data属性。

这不是一种好的设计。更好的方式是由DataStore提供一系列用于添加、删除和检索数据的公共接口，同时这些接口的运行方式不被外界探知。

所以，创建自定义DataStore类型的第二步就是提供一个用于与数据交互的方法，这些方法是其他组件和DataStore实例交互的方式。要达到这个目的，需要使用一个强大的JavaScript函数特征：**原型**属性。

函数在JavaScript中也同样是对象，这意味着他们有属性。在JavaScript中，所有通过构造函数创建的实例都可以访问其属性和方法的共享仓库：构造函数的prototype属性。

要创建实例，需要使用new关键字调用构造函数。new关键字不仅仅创建并返回实例，还在实例和构造函数的prototype属性间建立了一个特别的链接。只要是使用new关键字调用构造函数创建的实例就会有这个链接。

当我们为prototype添加了一个函数作为属性后，每个通过这个构造函数创建的实例都可以访问这个函数。可以在函数内部使用this关键字，它会指向当前实例。

为了进一步探究它的效果，在datastore.js中的原型上添加add函数。这时候可以把console.log删除。

```
(function (window) {
  'use strict';
  var App = window.App || {};

  function DataStore() {
    console.log('running the DataStore function');
    this.data = {};
  }

  DataStore.prototype.add = function (key, val) {
    this.data[key] = val;
  };

  App.DataStore = DataStore;
  window.App = App;
})(window);
```

在上述代码中，我们为DataStore.prototype添加了add属性，并赋值为一个函数。这个函数接受两个参数：key和val。在函数的内部，我们使用这两个参数来修改当前实例的data属性。

那么DataStore怎么存储咖啡订单呢？它会通过用户的邮箱地址（key）存储订单信息（val）。

DataStore足以支撑CoffeeRun的正常运行，所以不需要创建一个真正的数据库。它可以通过独立的标识符key存储信息val。因为使用了JavaScript对象，所以每个key都可以被视为数据库中的唯一条目。（在JavaScript对象中，属性名总是唯一的，就像全局命名空间中的函数名一样。如果尝试用相同的键名存储不同的数值，之前存储的数据会被覆盖。）

JavaScript对象的这个特性满足了所有数据库的一大需求：保持单独的数据片段。

保存代码，切换到浏览器，在控制台中创建一个DataStore实例，再通过add方法存储一些信息。

```
var ds = new App.DataStore();
ds.add('email', 'q@bond.com');
ds.add('order', 'triple espresso');
ds.data;
```

查看data属性，确认它们是否生效（如图8-9所示）。

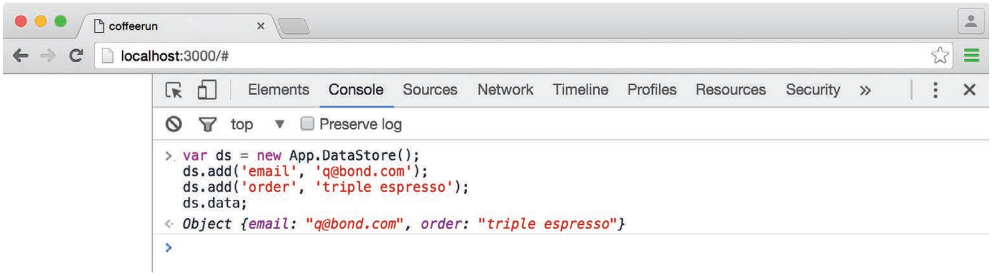


图8-9　调用一个原型方法

8.5.2　为构造函数添加方法

接下来，创建读取数据的方法。在datastore.js中添加一个用于查询某个特定键名的对应键值的方法和一个查询所有的键值对的方法。

```
...
  DataStore.prototype.add = function (key, val) {
    this.data[key] = val;
  };

  DataStore.prototype.get = function (key) {
    return this.data[key];
  };

  DataStore.prototype.getAll = function () {
    return this.data;
  };

  App.DataStore = DataStore;
  window.App = App;
})(window);
```

我们创建了一个get方法，该方法接受参数key，并在相应实例的data属性中查找键值。同时，我们还创建了一个getAll方法。尽管和get看起来很像，但getall会直接返回data属性的的引用。

现在可以通过DataStore实例添加和检索数据了。为了保持逻辑链的完整性，我们还需要添加一个删除信息的方法。现在就动手吧！

```
...
  DataStore.prototype.getAll = function () {
    return this.data;
  };

  DataStore.prototype.remove = function (key) {
    delete this.data[key];
```

```
  };

  App.DataStore = DataStore;
  window.App = App;
})(window);
```

当调用remove方法时，delete运算符会将一个键值对从对象上删除。

至此为止，我们已经完成了CoffeeRun中最重要的模块DataStore，它可以存储数据、根据查询需求提供相应的数据，并且按照指令删除不必要的数据。

要查看方法的效果，先保存代码，然后在browser-sync重新加载页面后切换到控制台。接着输入以下代码，它们可以测试DataStore中的方法。

```
var ds = new App.DataStore();
ds.add('m@bond.com', 'tea');
ds.add('james@bond.com', 'eshpressho');
ds.getAll();
ds.remove('james@bond.com');
ds.getAll();
ds.get('m@bond.com');
ds.get('james@bond.com');
```

如图8-10所示，DataStore实例现在能正常工作了。这些方法正是其他模块与应用数据库进行交互的方法。

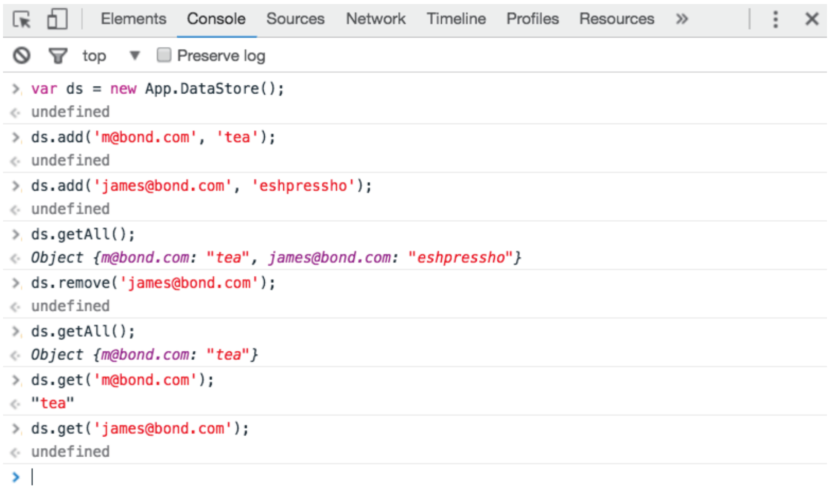


图8-10　使用原型上的方法与DataStore交流

下一个模块会使用相同的架构——一个接受参数并且修改其命名空间的IIFE——但是它会提供和DataStore完全不一样的功能。

8.6 创建 Truck 模块

接下来要编写的是Truck模块，它负责提供用于管理美食车的功能，比如创建、交付订单，打印等待中的订单列表。图8-11展示了Truck模块如何与DataStore模块协作。

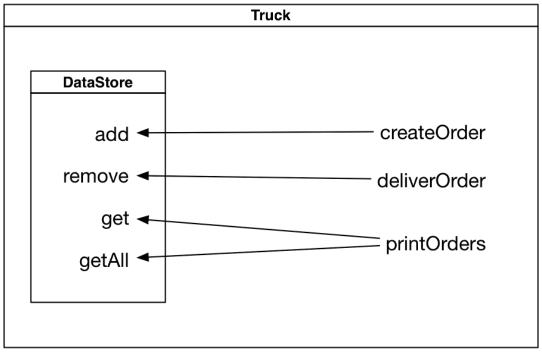


图8-11　Truck模块和DataStore模块进行交互

当Truck实例被创建后，它会被赋予一个DataStore对象。Truck对象拥有操作咖啡订单的功能，但是它不应该负责订单的存储和管理，而只需要将这些任务交给DataStore对象来完成就可以了。例如，当调用Truck的createOrder方法时，它只是调用DataStore的add方法。

创建scripts/truck.js文件并且在index.html中添加一个<script>标签。

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>coffeerun</title>
  </head>
  <body>
    <script src="scripts/datastore.js" charset="utf-8"></script>
    <script src="scripts/truck.js" charset="utf-8"></script>
  </body>
</html>
```

保存index.html文件。在truck.js中使用IIFE和构造函数建立一个Truck类型。

```
(function (window) {
  'use strict';
  var App = window.App || {};

  function Truck() {
  }

  App.Truck = Truck;
  window.App = App;

})(window);
```

接下来要为构造函数添加参数，这样才可以让每一个实例拥有独立的标识符和相应的DataStore实例。标识符是用于区别不同Truck的名字。DataStore实例在此时会扮演一个重要角色。

在truck.js中添加一个新参数。

```
(function (window) {
  'use strict';
  var App = window.App || {};

  function Truck(truckId, db) {
    this.truckId = truckId;
    this.db = db;
  }

  App.Truck = Truck;
  window.App = App;

})(window);
```

我们声明了truckId和db这两个参数，然后将它们作为属性分别分配给新建的实例。

下一步会为Truck实例添加管理咖啡订单的方法。订单数据包括一个邮箱地址和一个饮品的文字描述。

8.6.1　添加订单

第一个要添加的方法是createOrder。当这个方法被调用后，Truck实例将通过先前声明的DataStore实例的方法和db属性交互。具体来说，就是调用DataStore的add方法来存储咖啡订单，并且使用邮箱地址来关联订单。

在truck.js中声明新属性。

```
...
  function Truck(truckId, db) {
    this.truckId = truckId;
    this.db = db;
  }

  Truck.prototype.createOrder = function (order) {
    console.log('Adding order for ' + order.emailAddress);
    this.db.add(order.emailAddress, order);
  };

  App.Truck = Truck;
  window.App = App;

})(window);
```

在createOrder函数中输出消息到控制台，然后使用db属性的add方法来存储订单信息。

调用add方法十分简单，只需要指向Truck的db实例，然后调用它的add方法即可。无须在这

个文件里指定App.DataStore的命名空间或者在模块内提起DataStore的构造函数。只要一个对象拥有和Datastore相同的方法，它就可以被Truck使用，而Truck并不关心它们是怎么实现的。

保存文件，然后在控制台输入以下条目来测试createOrder方法：

```
var myTruck = new App.Truck('007', new App.DataStore());
myTruck.createOrder({ emailAddress: 'dr@no.com', coffee: 'decaf'});
myTruck.createOrder({ emailAddress: 'me@goldfinger.com', coffee: 'double mocha'});
myTruck.createOrder({ emailAddress: 'm@bond.com', coffee: 'earl grey'});
myTruck.db;
```

得到的结果应该和图8-12相似。

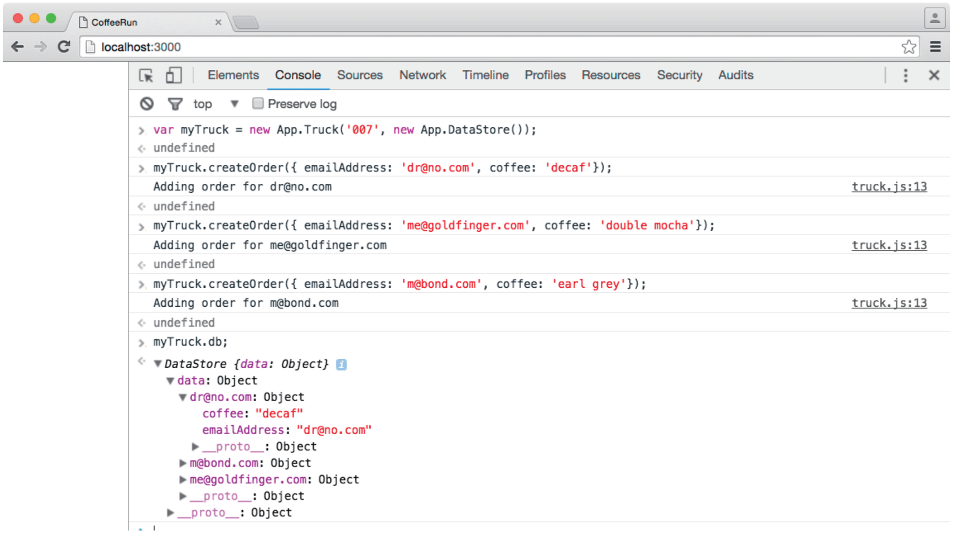


图8-12　测试Truck.prototype.createOrder

当控制台输出myTruck.db的值时，点击▶图标来查看嵌套的属性（如data对象里的dr@no.com属性）。

8.6.2　删除订单

当订单交付后，Truck应该从数据库中删除相应的订单，所以需要为truck.js的Truck.prototype对象添加deliverOrder方法。

```
...
  Truck.prototype.createOrder = function (order) {
    console.log('Adding order for ' + data.emailAddress);
    this.db.add(data.emailAddress, order);
  };

  Truck.prototype.deliverOrder = function (customerId) {
    console.log('Delivering order for ' + customerId);
    this.db.remove(customerId);
```

```
    };

    App.Truck = Truck;
    window.App = App;

})(window);
```

我们赋予Truck.prototype.deliverOrder一个函数表达式。这个函数接受一个customerId参数，随后它会被传递给this.db.remove，它的值应该是一个与订单关联的邮箱地址。

和createOrder一样，deliverOrder也只调用this.db的remove方法，而无须知道remove方法如何运行。

保存文件，切换到控制台。创建一个Truck实例，通过createOrder方法添加一些订单，接着检测deliverOrder是否成功删除订单。（你可以在调用createOrder和deliverOrder函数后敲击回车键或者使用Shift + Enter，但是要记得在每个myTruck.db语句后敲击回车键。）

```
var myTruck = new App.Truck('007', new App.DataStore());
myTruck.createOrder({ emailAddress: 'm@bond.com', coffee: 'earl grey'});
myTruck.createOrder({ emailAddress: 'dr@no.com', coffee: 'decaf'});
myTruck.createOrder({ emailAddress: 'me@goldfinger.com', coffee: 'double mocha'});
myTruck.db;
myTruck.deliverOrder('m@bond.com');
myTruck.deliverOrder('dr@no.com');
myTruck.db;
```

输入以上指令后，myTruck.db的订单会在调用deliverOrder后发生改变（如图8-13所示）。

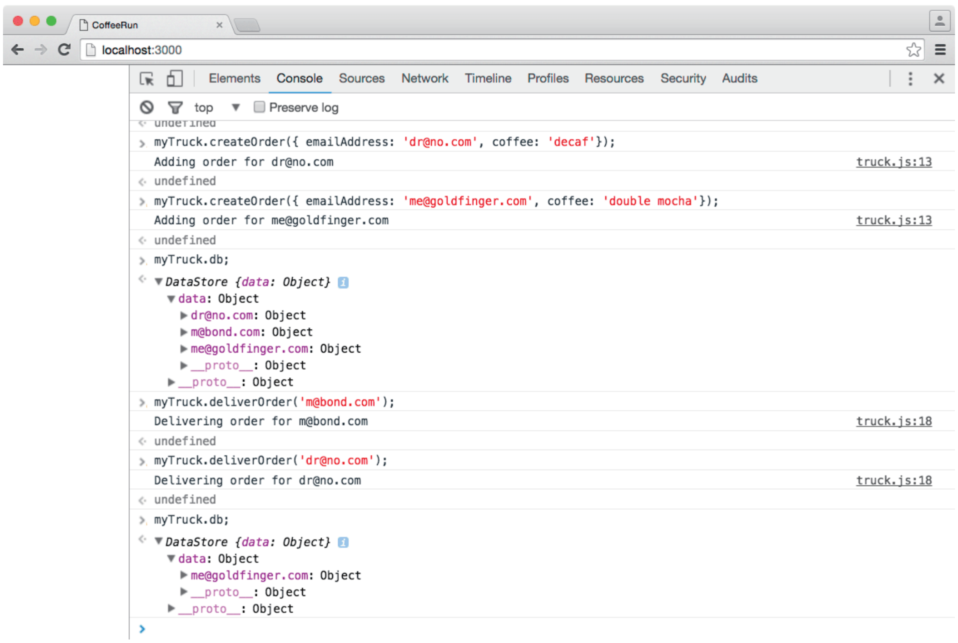


图8-13　通过Truck.prototype.deliverOrder删除订单数据

注意，控制台会在你点击▶图标时展示数据。如果你直到执行完所有`deliverOrder`函数后才查看`myTruck.db`，就会发现数据好像没有被添加过一样（如图8-14所示）。

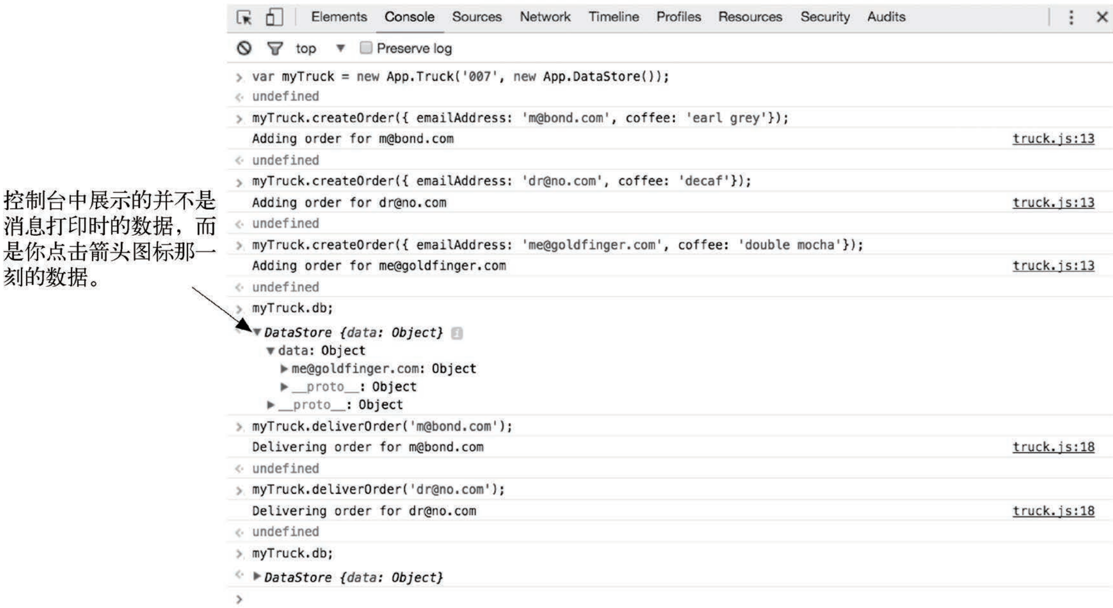


图8-14　通过点击箭头图标展示数据

8.7　调试

最后一个要加到`Truck.prototype`对象上的方法是`printOrders`。这个方法接受一个由客户邮箱地址组成的数组，遍历数组，然后使用`console.log`打印出订单信息。

这个方法的代码和之前写过的函数和方法非常相像。不过它会带来一个bug，通过Chrome的调试工具可以揪出它。

不用着急，一步一步来。首先在truck.js中添加一个`printOrders`的基本版本。在函数中通过`db`对象获取所有的咖啡订单，然后使用`Object.keys`方法来获取一个包含所有订单的邮箱地址的数组，最后遍历这个数组，为数组里的每个函数调用一次回调函数。

```
...
  Truck.prototype.deliverOrder = function (customerId) {
    console.log('Delivering order for ' + customerId);
    this.db.remove(customerId);
  };

  Truck.prototype.printOrders = function () {
    var customerIdArray = Object.keys(this.db.getAll());
```

```
        console.log('Truck #' + this.truckId + ' has pending orders:');
        customerIdArray.forEach(function (id) {
          console.log(this.db.get(id));
        });
      };

      App.Truck = Truck;
      window.App = App;

    })(window);
```

在新的printOrders方法中调用this.db.getAll来获取所有订单的键值对，然后将它们传入Object.keys，从而获得一个仅包含键名的数组，再将这个数组赋值给变量customerIdArray。

在遍历这个数组时，将一个回调函数传给forEach。在回调函数内，尝试通过一个id（客户的邮箱地址）来获取订单信息。

保存文件，再回到控制台。创建一个Truck实例，然后添加一些咖啡订单，尝试使用新的printOrders方法。

```
var myTruck = new App.Truck('007', new App.DataStore());
myTruck.createOrder({ emailAddress: 'm@bond.com', coffee: 'earl grey'});
myTruck.createOrder({ emailAddress: 'dr@no.com', coffee: 'decaf'});
myTruck.createOrder({ emailAddress: 'me@goldfinger.com', coffee: 'double mocha'});
myTruck.printOrders();
```

然而我们并没有看到一组咖啡订单，反倒看到了错误提示Uncaught TypeError: Cannot read property 'db' of undefined（如图8-15所示）。

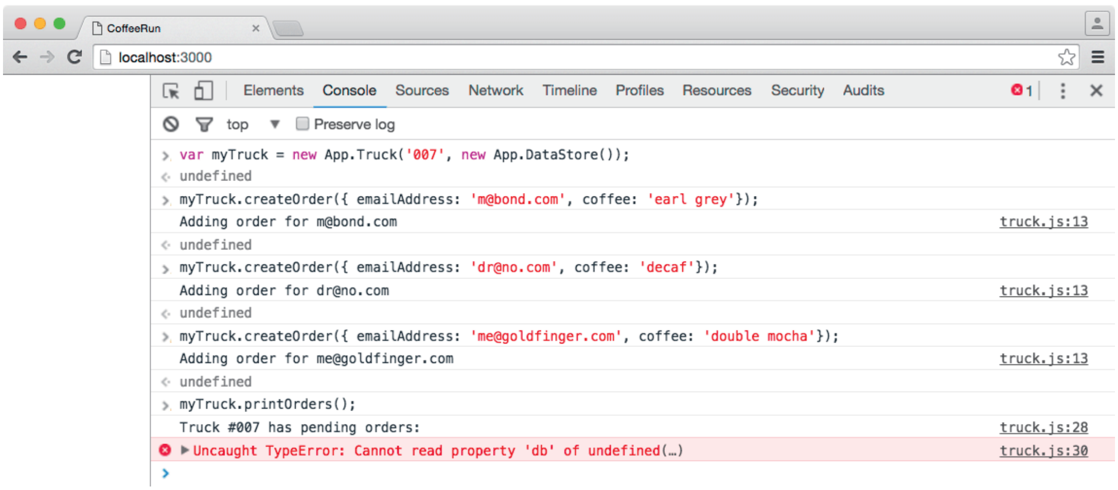


图8-15　运行printOrders时抛出的错误

这是在编写JavaScript时最容易见到的错误之一。许多开发者都为此抓狂，因为它很难定位。但是只要知道怎么使用调试工具，就能快速定位问题，而这也是接下来我们将要学习的。

8.7.1　使用开发者工具定位bug

调试需要我们通过重现错误定位问题，而Chrome调试工具让这个过程变得十分轻松。

当出现一个错误时，控制台会显示错误所在的文件名和行号。（在图8-15中显示为truck.js:30;，你看到的行号可能略有不同。）单击该文本，在调试工具中打开这行讨厌的代码（如图8-16所示）。

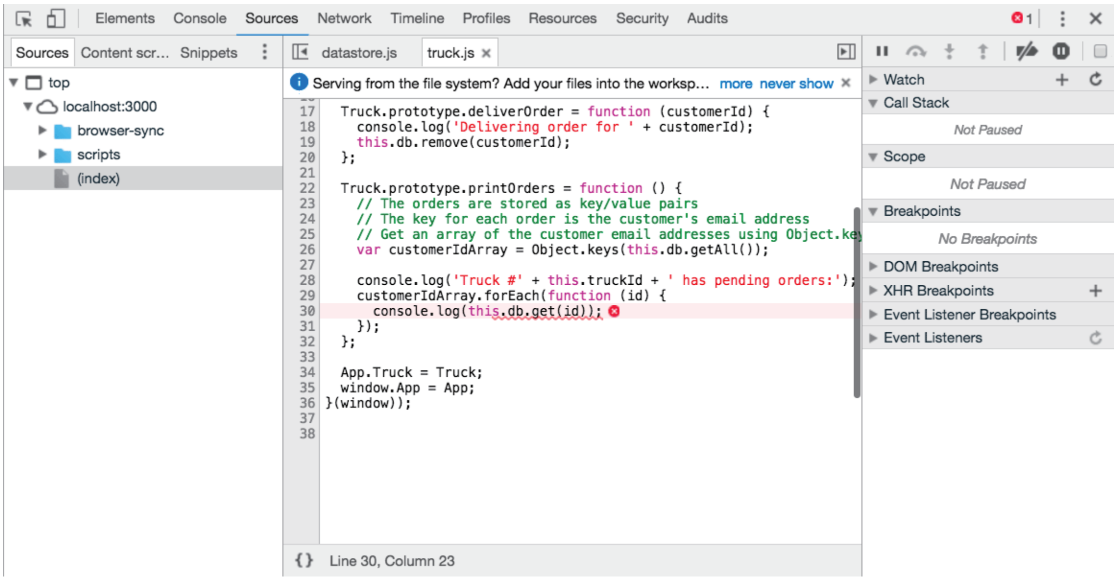


图8-16　在调试工具中查看错误

现在查看的是开发者工具的源码面板。点击红色图标来查看错误信息（如图8-17所示）。

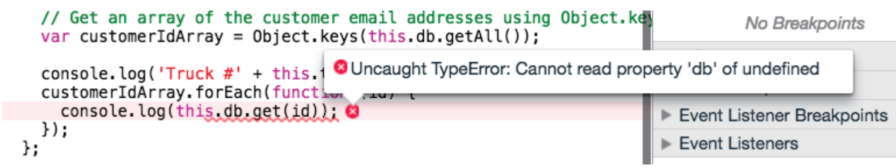


图8-17　在源码面板中标示的错误行

这个错误信息表示浏览器认为你要从一个不存在的对象上读取名为db的属性。

下一步是运行代码直到错误行，然后检查该对象的值。在源码面板上点击左边的行号，这样就会在调试器上添加一个断点，告知浏览器运行到该行时暂停。设置断点后，行号会变为蓝色，而在右侧的断点面板上会出现一个新条目（如图8-18所示）。

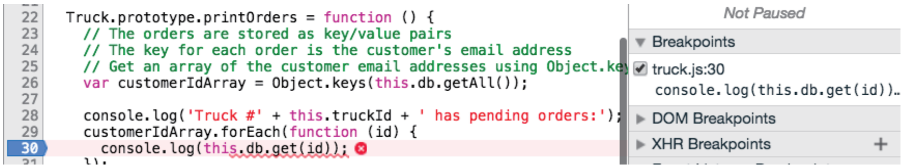


图8-18　设置断点

按下Esc键，调出控制台（如图8-19所示）。控制台也被称作抽屉，我们可以在查看源码面板的同时，使用控制台进行交互。

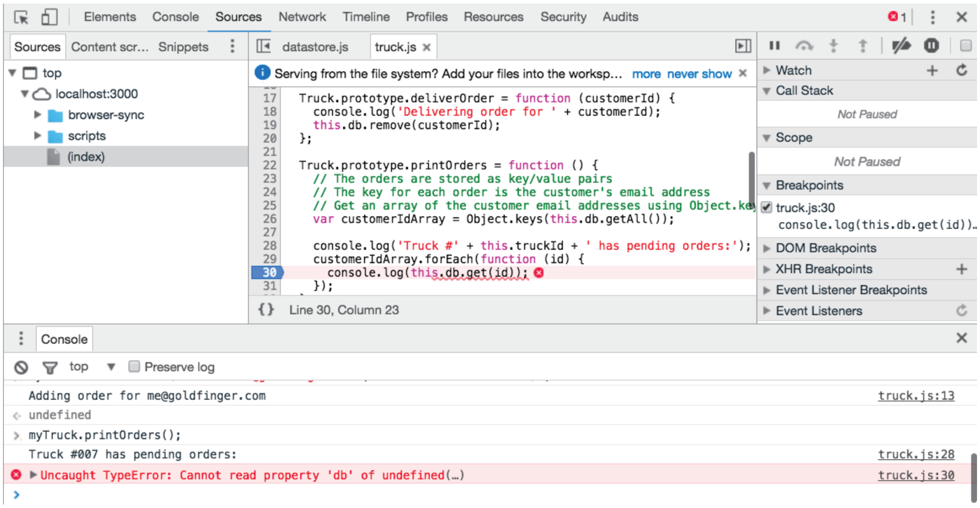


图8-19　展示控制台抽屉

在控制台中再次运行`myTruck.printOrders();`。这时浏览器会激活调试器，代码会停在断点处（如图8-20所示）。

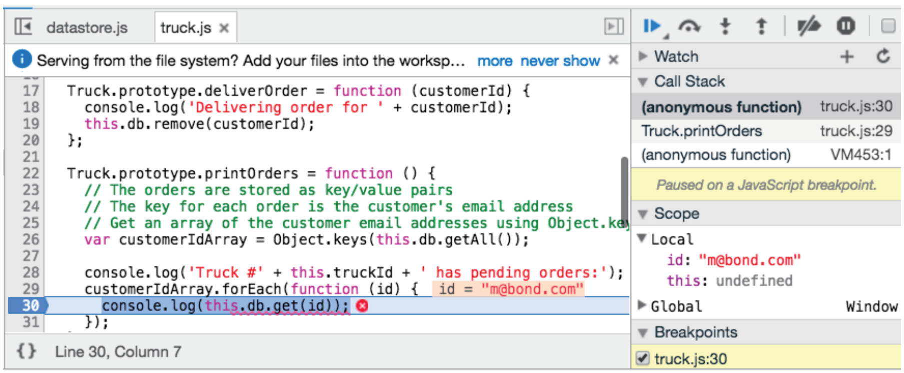


图8-20　调试器暂停在断点处

当调试器暂停后，我们就有权限访问当时所能获得的变量。此时，可以使用控制台检测变量的值，从而查找问题的蛛丝马迹。

我们要通过执行错误行来重现错误。首先从最里层的括号开始，也就是这里的id变量。当我们在控制台中输入它的时候，会得到m@bond.com值（如图8-21所示）。

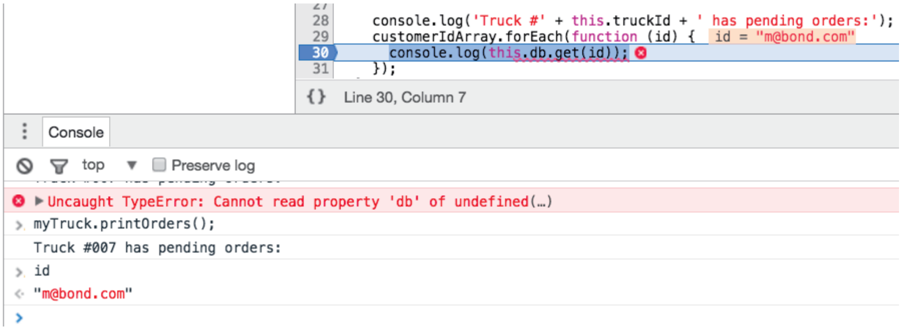


图8-21　检查最内层的值

因为这时并没有出现错误，所以继续尝试检查括号外的代码，this.db.get(id)。在控制台上执行代码，代码会出现错误（如图8-22所示）。

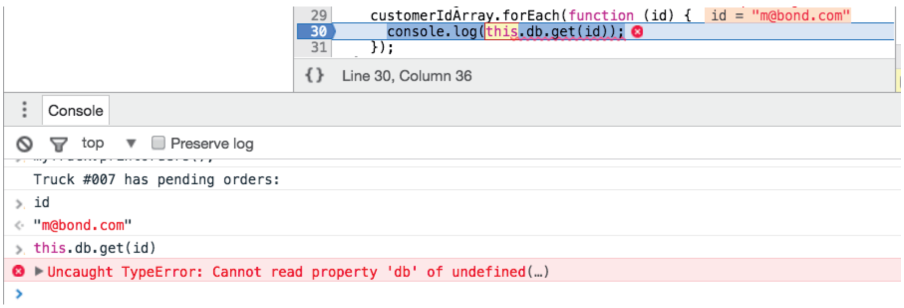


图8-22　重现错误

现在可以进一步定位问题了。从右侧开始，尝试删除部分代码片段然后执行，直到错误不再重现。先输入this.db.get，再输入this.db。控制台依旧报错（如图8-23所示）。

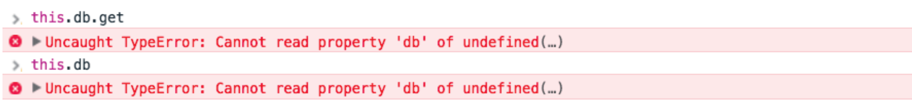


图8-23　继续搜寻

最后，输入this，终于没有错误抛出了（如图8-24所示）。

8

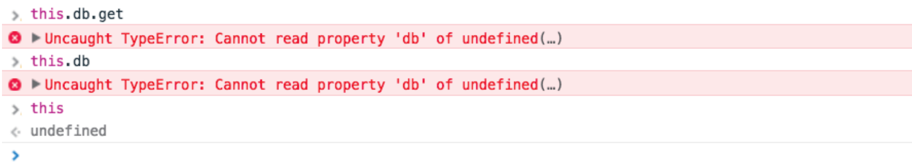


图8-24　修剪代码以找出错误原因

为什么this在回调函数内的值是undefined呢？因为this在回调函数里并没有被赋值，所以我们需要为其赋值。

这个情形和Truck.prototype方法有点区别。在原型的方法中，this指代Truck的实例。尽管回调函数在Truck.prototype.printOrder中，但是它拥有自己的this变量。而这个变量并未被赋值，所以为undefined。

在修改代码前，我们还需要熟悉一下另外两种定位错误的方法。如果将鼠标悬停在源码面板中的某一部分代码上，调试器会显示它们的值——将鼠标悬停在this上，会看到值undefined（如图8-25所示）。

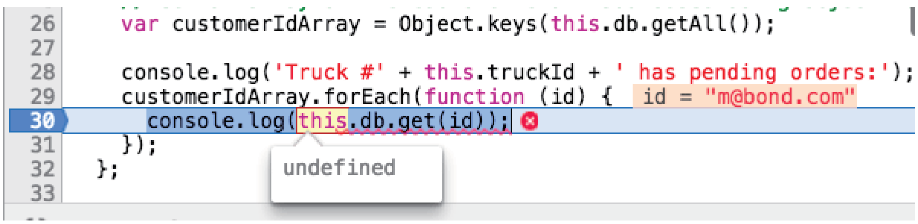


图8-25　悬停鼠标来查看数值

而在右侧的Scope面板中有一个变量列表，在其中可以看到id和this，this的值自然也是undefined（如图8-26所示）。

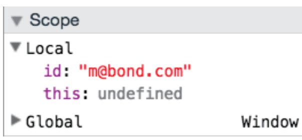


图8-26　在Scope面板上查看变量

点击右上角的蓝色▶按钮（如图8-27所示），它会让代码再次执行。



图8-27　调试器控制面板

最后，再次点击蓝色行号以移除断点（如图8-28所示）。



图8-28　点击行号，移除断点

现在去搞定那个麻烦的bug吧。

8.7.2　使用bind设置this

在JavaScript中，函数内的关键字this会在函数调用时被自动赋值。对于构造函数和原型上的方法来说，它们的this是相应对象的实例。这个实例被称作函数调用的**所有者**，而this让我们有权限去访问所有者的属性。

如先前所说，回调函数并没有将对象自动分配给this。你可以使用函数的bind方法来指定函数的所有者。（记住，JavaScript的函数本质上也是对象，所以它们拥有自己对应的属性和方法，比如bind。）

bind方法接受一个对象实参并返回一个新版本函数。当调用这个新版本函数时，它会在函数内部将对象实参绑定到this上。

而在forEach的回调函数中，因为它没有所有者，所以this的值是undefined。可以通过调用bind函数并且传入Truck实例进行修复。

在truck.js中调用bind。

```
...
  Truck.prototype.printOrders = function () {
    var customerIdArray = Object.keys(this.db.getAll());

    console.log('Truck #' + this.truckId + ' has pending orders:');
    customerIdArray.forEach(function (id) {
      console.log(this.db.get(id));
    }.bind(this));
  };
...
```

在forEach的回调函数外，this指代Truck实例。在forEach括号内部的匿名函数后马上调用.bind(this)方法，从而将修改过的匿名函数传给forEach。这个修改过的函数的所有者是Truck实例。

保存并且确认订单是否正确地被打印。可能需要重新声明myTruck并再次运行createOrder。输出应如图8-29所示。

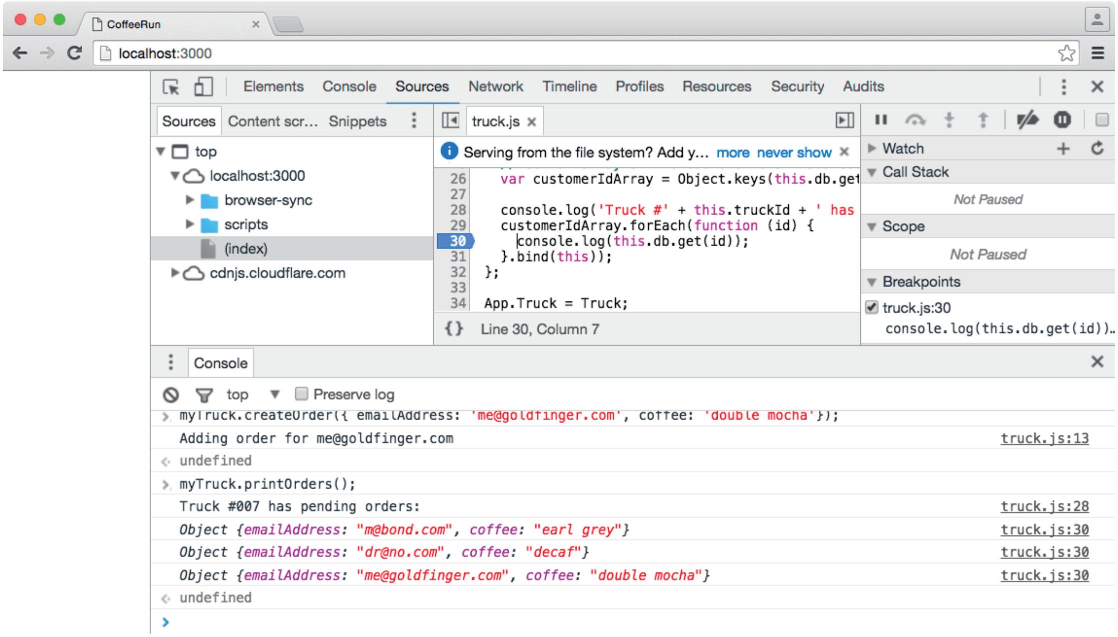


图8-29　使用`bind(this)`让`printOrders`正常工作

8.8　在页面加载时初始化 CoffeeRun

现在`DataStore`和`Truck`模块都能正常工作了。可以通过控制台对`Truck`进行实例化操作，并为它分配一个`DataStore`实例。

现在我们将创建一个模块，让它在页面加载时自动执行上述步骤。创建一个scripts/main.js文件，在index.html上添加对应的`<script>`标签。

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>coffeerun</title>
  </head>
  <body>
    <script src="scripts/datastore.js" charset="utf-8"></script>
    <script src="scripts/truck.js" charset="utf-8"></script>
    <script src="scripts/main.js" charset="utf-8"></script>
  </body>
</html>
```

保存index.html，然后在main.js上添加IIFE，就像在其他模块中所做的一样，但是这一次不需要为`window.App`添加新属性。建立main.js如下：

```
(function (window) {
  'use strict';
  var App = window.App;
  var Truck = App.Truck;
  var DataStore = App.DataStore;
})(window);
```

这个模块会接受window对象并且在函数内部使用，它同样会检索我们在window.App命名空间下定义的构造函数。

理论上来说，可以使用完整的名称（如App.Truck和App.DataStore），不过更短的名字会增加代码的可读性。

创建Truck实例

然后就像刚刚在控制台中所做的一样，创建一个Truck实例，并且为其提供一个id和一个新的DataStore实例。

在main.js中调用Truck的构造函数，并传入id ncc-1701和DataStore实例。

```
(function (window) {
  'use strict';
  var App = window.App;
  var Truck = App.Truck;
  var DataStore = App.DataStore;
  var myTruck = new Truck('ncc-1701', new DataStore());
})(window);
```

这基本和我们先前在控制台中输入的代码一样，但是不需要在Truck或者DataStore前使用App前缀了，因为我们已经分别创建了指向App.Truck和App.Datastore的本地变量。

这时的应用代码已经基本完整了。然而，我们还是不能和Truck实例交互。为什么呢？因为变量是在main模块的函数内部声明的，而函数外部（包括控制台）无法访问函数内部的变量。

所以，为了便于交互，需要在main.js中将Truck暴露到全局命名空间。

```
(function (window) {
  'use strict';
  var App = window.App;
  var Truck = App.Truck;
  var DataStore = App.DataStore;
  var myTruck = new Truck('ncc-1701', new DataStore());
  window.myTruck = myTruck;
})(window);
```

保存代码，然后再切至控制台。手动刷新页面，确保之前的代码已经被清除。

输入myTruck，此时会发现控制台正在尝试进行自动补全（如图8-30所示），这意味着它已经发现通过window对象暴露的myTruck变量了。

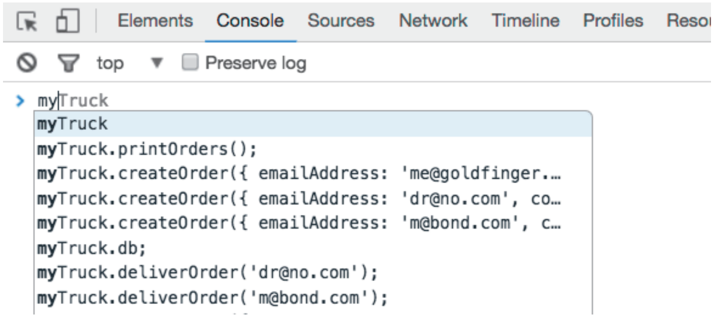


图8-30　控制台在全局命名空间下发现了myTruck

接下来尝试多次调用`myTruck.createOrder`，并且传入一些测试数据。这时候，控制台会根据输入历史自动补全`createOrder`的调用语句，这大大减轻了你的工作量（如图8-31所示）。

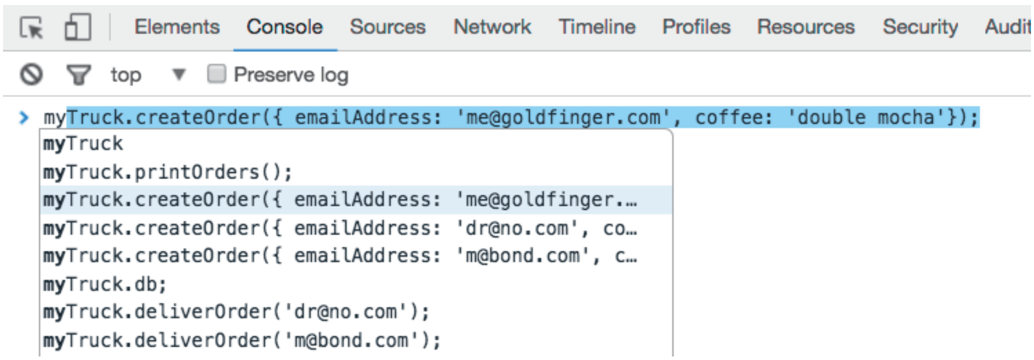


图8-31　控制台利用输入历史自动补全createOrder调用

也可以输入以下代码来进行测试。

```
myTruck.createOrder({ emailAddress: 'me@goldfinger.com', coffee: 'double mocha'});
myTruck.createOrder({ emailAddress: 'dr@no.com', coffee: 'decaf'});
myTruck.createOrder({ emailAddress: 'm@bond.com', coffee: 'earl grey'});
myTruck.printOrders();
myTruck.deliverOrder('dr@no.com');
myTruck.deliverOrder('m@bond.com');
myTruck.printOrders();
```

在执行了`createOrder`、`printOrders`和`deliverOrder`方法后，可以看到如图8-32所示的结果。

恭喜！你已经完成了CoffeeRun的底层架构。它暂时还没有UI结构，不过我们会在下一章为其添加。那时候也无须在核心代码上进行修改了，因为UI层只会调用`Truck.prototype`上的方法。

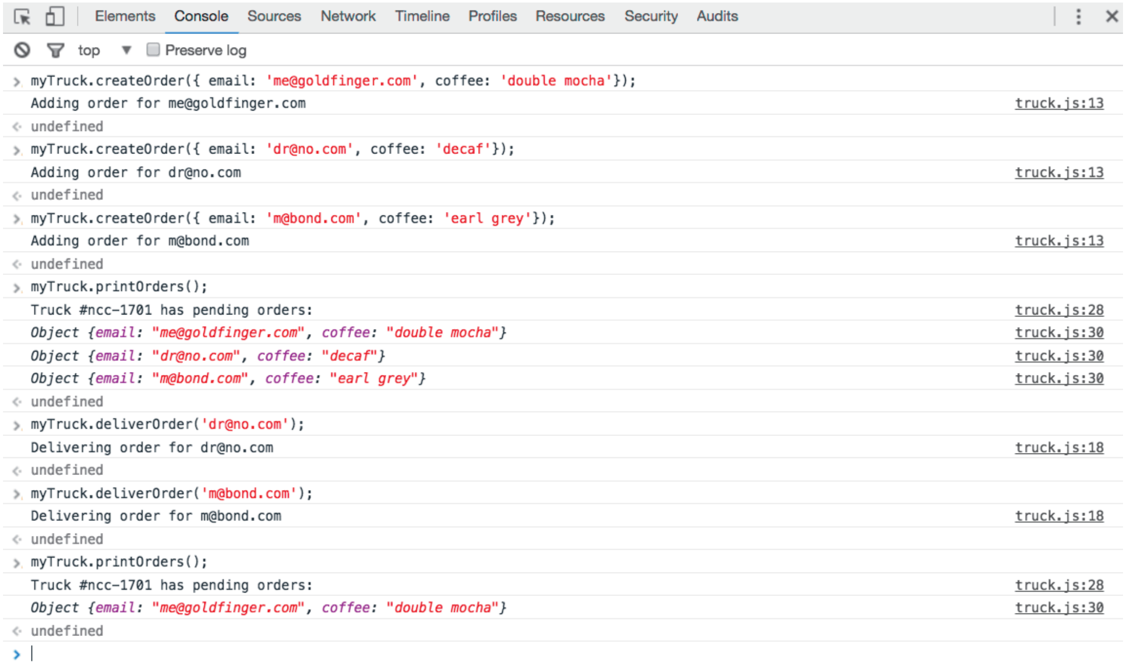


图8-32　一个繁忙的美食车

这就是模块化的好处：你可以分层处理应用，而每一个新层级都基于旧模块搭建而成。

8.9　初级挑战：使用非星迷熟悉的餐车 ID[①]

在main.js中传入不同的字符串作为`truckId`。

（有一些比较好的选项，例如Serenity、KITT或者Galactica。HAL或许不是一个好的选择。[②]）

8.10　延展阅读：模块私有数据

在模块中，构造函数和原型方法可以访问IIFE内部声明的所有变量。不过也可以将一部分方法添加到原型上，这样既可以在实例间分享数据，又可以对模块外部隐藏数据。例如：

```
(function (window) {
  'use strict';
  var App = window.App || {};
  var launchCount = 0;
```

① ncc-1701是《星际迷航》中的主角船联邦星舰企业号的代号。——译者注

② Serenity为电影《冲出宁静号》中的飞船，KITT为电视剧《霹雳游侠》中的智能跑车，Galactica为电视剧《太空堡垒卡拉狄加》中的飞船，HAL为《2001太空漫游》中的智能电脑。——译者注

```
  function Spaceship() {
    // 在此处插入初始化代码
  }

  Spaceship.prototype.blastoff = function () {
    // 闭包允许我们访问 launchCount属性
    launchCount++;
    console.log('Spaceship launched!')
  }

  Spaceship.prototype.reportLaunchCount = function () {
    console.log('Total number of launches: ' + launchCount);
  }

  App.Spaceship = Spaceship
  window.App = App;
})(window);
```

其他语言会提供一种方法来声明**私有**变量，不过JavaScript没有这种方法。但我们可以使用闭包（使用外部声明的变量的函数）来模拟私有变量。

8.11　中级挑战：私有化数据

更新DataStore让data属性私有化。

有任何理由**拒绝**做这件事吗？想像一下如果声明了多个DataStore实例会发生什么？

8.12　延展阅读：在 forEach 的回调函数中设置 this

我之前撒了个小谎，bind并不是为forEach的回调函数设置this值的**唯一**方法。

通过在MDN（developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach）上查看关于Array.prototype.forEach的文档，可知forEach接受可选的第二个参数，该参数会被当作回调函数中的this。

这意味着也可以这么编写printOrders：

```
...
Truck.prototype.printOrders = function () {
  var customerIdArray = Object.keys(this.db.getAll());

  console.log('Truck #' + this.truckId + ' has pending orders:');
  customerIdArray.forEach(function (id) {
    console.log(this.db.get(id));
  }, this);
};
...
```

但bind仍然是一个十分有用的函数，在接下来的几章中还会看到它。Truck.prototype.printOrders为学习这些语法提供了一个很好的机会。

第 9 章

Bootstrap简介

9

本章将为UI创建HTML标记。我们会采用流行的Bootstrap CSS框架提供的样式来美化UI，这样就不必自己创建CSS，从而能专注于JavaScript应用程序逻辑——这部分内容将在第10章中学习。

我们将分两步来为CoffeeRun应用程序创建UI。首先创建一个表单，用户可以在其中输入包含所有细节的咖啡订单（如图9-1所示）。接着，搭建一个用于显示现有咖啡订单的清单。每个部分都会有相应的JavaScript模块处理用户交互。

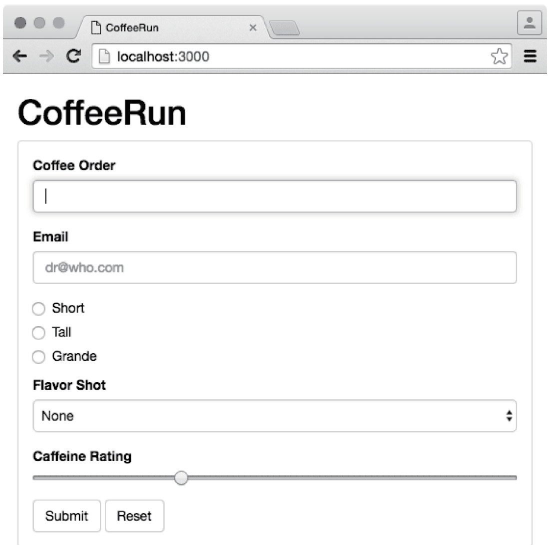


图9-1　使用Bootstrap样式的CoffeeRun

9.1　添加 Bootstrap

Bootstrap CSS类库提供了一系列可以应用于网站和应用程序的样式。但是因为其太过于普及，所以如果不进行一点儿定制工作，你的网站很可能会和其他网站“撞衫”。不过Bootstrap确

实非常适合快速创建好看的原型。

和在Ottergram中引用normalize.css一样，你也可以从cdnjs.com加载Bootstrap。使用Bootstrap 3.3.6版本，地址是cdnjs.com/libraries/twitter-bootstrap/3.3.6。（如果你想在项目中使用最新版本，请到cdnjs.com搜索“twitter bootstrap”。）

确保得到的是bootstrap.min.css的链接（如图9-2所示），而不是其字体或者主题的链接。

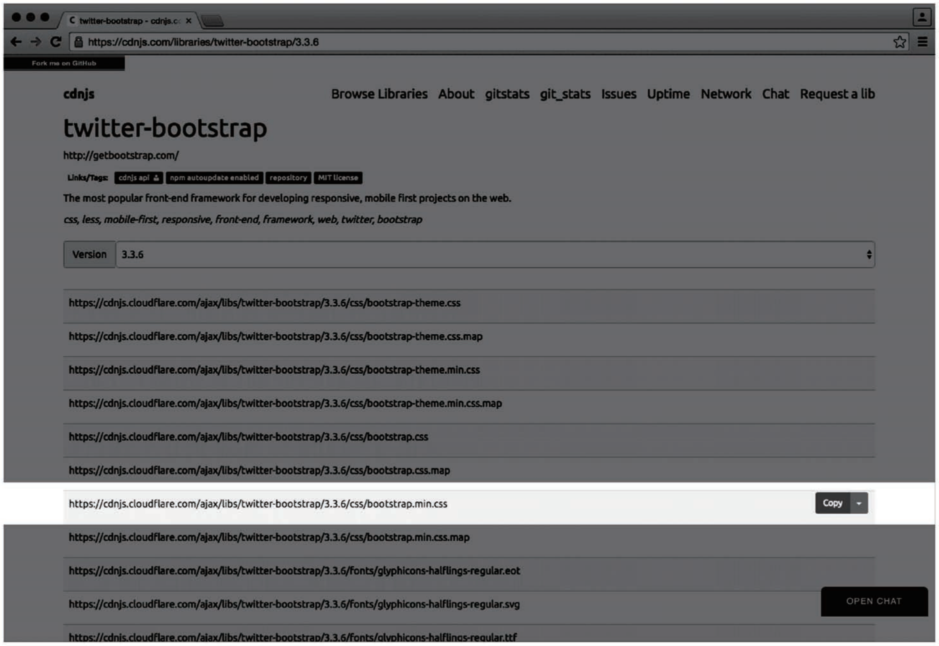


图9-2　cdnjs.com的twitter-bootstrap页面

复制链接之后，打开index.html添加一个`<link>`标签，把该标签的`href`属性设置为复制的链接地址。（为了适应版面，不得不将`href`属性换行，但是你应该把它写成一行。）

```
...
  <head>
    <meta charset="utf-8">
    <title>coffeerun</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootst
rap/3.3.6/css/bootstrap.min.css">
  </head>
...
```

Bootstrap的原理

Bootstrap可以为网站和Web应用提供“开箱即用”的自适应样式。大多数时候，只需要引入CSS文件，并给HTML标记添加类即可。我们将用到的一个重要的类是`container`类。

在index.html的`<body>`元素中添加`container`类，同时在里面添加一个页头。

```
...
      <title>coffeerun</title>
      <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootst
rap/3.3.6/css/bootstrap.min.css">
  </head>
  <body>
  <body class="container">
    <header>
      <h1>CoffeeRun</h1>
    </header>
    <script src="scripts/datastore.js" charset="utf-8"></script>
    <script src="scripts/truck.js" charset="utf-8"></script>
    <script src="scripts/main.js" charset="utf-8"></script>
  </body>
</html>
```

`container`类用于包裹所有需要适应视口大小的内容，它提供了基本的布局自适应功能。

保存index.html，确保 browser-sync进程正在运行，查看网页。现在它看起来应该如图9-3所示。

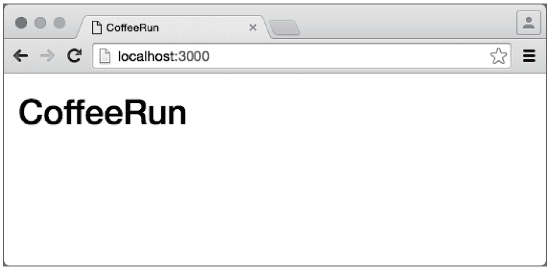


图9-3　具有Bootstrap样式的页头

虽然页面上还没有太多内容，但是页头已经有了一个令人舒服的内边距，并且也有了字体样式。

Bootstrap具有大量不同视觉元素的样式，CoffeeRun只会接触到其皮毛，后面的章节会探索更多的样式。现在，是时候为订单表单添加标记了。

9.2　创建订单表单

在index.html中新创建的`<header>`元素的下方添加一个`<section>`标签、两个`<div>`标签和一个`<form>`标签。

```
...
    <header>
      <h1>CoffeeRun</h1>
    </header>
    <section>
      <div class="panel panel-default">
        <div class="panel-body">
```

```
          <form data-coffee-order="form">
            <!-- Input elements will go here -->
          </form>
        </div>
      </div>
    </section>
    <script src="scripts/datastore.js" charset="utf-8"></script>
...
```

<form>标签将是所有重要交互的发生地，给它添加一个值为form的data-coffee-order属性。就像在Ottergram中所做的一样，我们也将在CoffeeRun中使用数据属性以便于Javascript访问DOM元素。

为了页面布局，添加两个<div>标签。其实<div>标签并不是那么重要，重要的是为<div>添加了panel、panel-default和panel-body类，这些类都将是触发样式改变的Bootstrap类。

切记，<div>只是用于包裹其他标记的通用块级容器。这些<div>标签会与包裹它们的父级元素拥有相同的水平宽度。它们将在CoffeeRun中被频繁使用，我们也经常会在Bootstrap文档中看到它们的身影。

你可能好奇为什么<section>标签包裹了<div>和<form>。那是因为<div>并没有语义，而<section>有，它可以对其他标记按逻辑进行分组。这里的<section>把表单的UI给框起来了。你可以很轻易地为页面创建另一个<section>，用来包裹其他UI部分。

9.2.1　添加文本输入字段

咖啡订单才是我们关心的重要信息。如果过去十年里你一直在咖啡店工作，就会知道咖啡订单可能会相当复杂。现在先使用一个单行文本字段来填写订单，稍后还会添加更多的字段来获取更多信息。

当对表单使用Bootstrap时，可以添加额外的<div>元素，这些元素仅仅是为了应用Bootstrap库中定义的样式。

向index.html添加一个类名为form-group的<div>。Bootstrap的form-group类为表单元素提供了一致的垂直间距。然后添加<label>和<input>元素。

```
...
        <div class="panel panel-default">
          <div class="panel-body">
            <form data-coffee-order="form">
              <输入框写在这里>
              <div class="form-group">
                <label>Coffee Order</label>
                <input class="form-control" name="coffee">
              </div>
            </form>
          </div>
        </div>
...
```

form-control是Bootstrap定义的另一个类，它能为表单元素提供布局和排版样式。

保存index.html，并在浏览器中检查结果（如图9-4所示）。

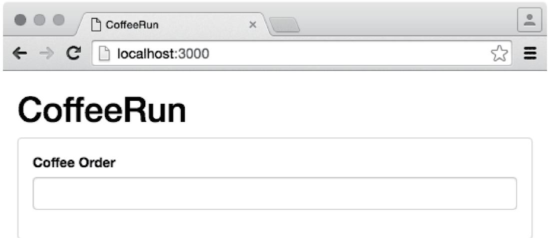


图9-4 咖啡订单的输入字段

<input>元素默认为单行文本字段。该元素除了有form-control类，还具有一个name属性。当提交表单的时候，数据将会被发送到服务器，这个name属性将与数据一起发送。如果将表单数据视为键值对，那么name属性对应的就是键名，用户在字段中输入的数据是键值。

1. 关联标签和表单元素

<label>标签可以大大增强表单元素的可用性。我们为<label>标签设置for属性来标记对应表单元素，for属性的值与要标记的表单元素的id属性相匹配。

在index.html中，分别为<label>和<input>表单元素添加for和id属性，并给这两个属性设置相同的coffeeOrder值。

```
...
        <div class="panel panel-default">
          <div class="panel-body">
            <form data-coffee-order="form">
              <div class="form-group">
                <label for="coffeeOrder">Coffee Order</label>
                <input class="form-control" name="coffee" id="coffeeOrder">
              </div>
            </form>
          </div>
        </div>
...
```

当一个<label>链接到了一个表单元素时，可以点击页面上的<label>文本，这样就会使其链接的表单元素处于激活状态。我们应该始终把<label>元素链接到其相应的表单元素。

想看结果的话，就保存index.html，切换到浏览器，然后单击咖啡订单的标签文本。此时，<input>应该获得了焦点，然后你就可以输入了（如图9-5所示）。

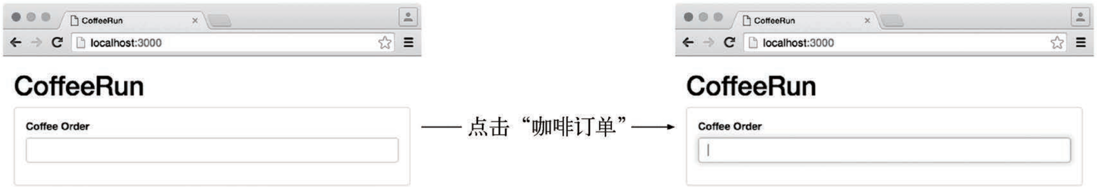


图9-5　单击关联的label，input获取焦点

2. 添加自动聚焦功能

因为这是屏幕上的第一个字段，所以我们希望用户在页面加载完成之后无须点击，就能立即在其中输入文本。

要实现这一点，请为index.html中的<input>添加一个autofocus属性。

```
...
            <div class="form-group">
              <label for="coffeeOrder">Coffee Order</label>
              <input class="form-control" name="coffee" id="coffeeOrder" autofocus>
            </div>
...
```

保存对index.html的更改，然后返回浏览器。你会看到文本输入字段在页面加载完成后被聚焦，且呈高亮显示。

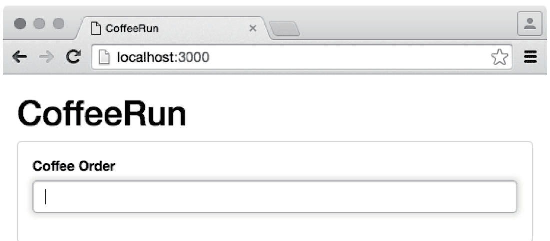


图9-6　页面加载完成后，拥有autofocus属性的文本字段

请注意，autofocus属性没有值，它也不需要有值。只要<input>标签中存在autofocus属性，浏览器就会知道要激活该字段。autofocus属性是**布尔**属性，这意味着它唯一可能的值是true或者false。你只需要将autofocus属性添加到标签，即可设置其值。当属性存在时，autofocus为true；当属性不存在时，autofocus为false。

3. 添加邮箱输入字段

在创建Trunk和DataStore模块时，可以通过客户的邮箱地址跟踪订单。现在，我们将使用

另一个<input>元素来获取该信息。

向index.html添加另一个包裹了<label>和<input>的.form-group元素。把<input>元素的type属性设置为email，将name属性设置为emailAddress，将id设置为emailInput。另外添加一个value属性，并把其值设置为空字符串，这保证此字段在加载网页时为空。最后使用id把<label>和<input>关联起来。

```
...
            <form data-coffee-order="form">
              <div class="form-group">
                <label for="coffeeOrder">Coffee Order</label>
                <input class="form-control" name="coffee" id="coffeeOrder" autofocus>
              </div>
              <div class="form-group">
                <label for="emailInput">Email</label>
                <input class="form-control" type="email" name="emailAddress"
                  id="emailInput" value="">
              </div>
            </form>
...
```

保存index.html，打开浏览器，查看新的表单字段（如图9-7所示）。

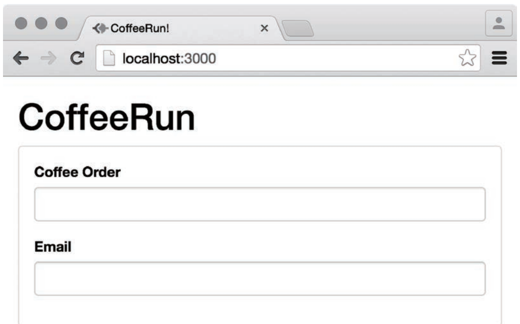


图9-7　邮箱地址输入字段

4. 用placeholder显示要输入的示例文本

有时候用户希望得到应该在文本字段中输入什么的建议。要创建示例文本，请使用placeholder属性。

在index.html中为新添加的<input>元素添加placeholder属性。

```
...
            <div class="form-group">
              <label for="emailInput">Email</label>
              <input class="form-control" type="email" name="emailAddress"
                id="emailInput" value="" placeholder="dr@who.com">
            </div>
...
```

保存文件，结果将如图9-8所示。

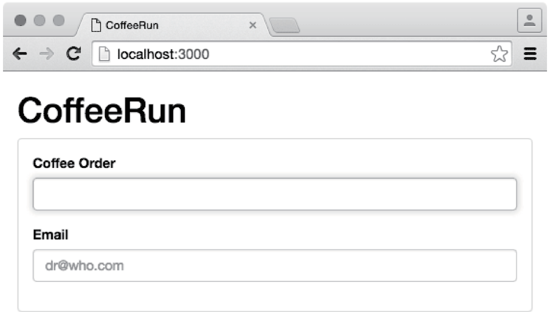


图9-8　邮箱文本框具有placeholder文本

placeholder的值将会在文本字段中显示，直到用户输入文本时才消失。如果用户删除字段中所有的文本，placeholder文本则会再次出现。

9.2.2　提供单选按钮

接下来，我们希望用户能够指定他们咖啡饮品的杯型——他们能够在小杯、大杯、超大杯中选择，并且只能选择一种。对于此类型的数据输入，可以使用type属性为radio的<input>字段。

单选按钮标记与其他<input>字段不同，每个单选按钮都包含被<label>元素**包裹**的<input>字段。<label>将被包含在一个类名**也**为radio的<div>中。

不需要对咖啡订单和邮箱地址的<label>定义for属性，因为<input>被<label>包裹，它们会被自动关联。

为什么单选按钮的HTML与众不同？那是因为Bootstrap以区别于其他表单元素的方式为单选按钮添加样式。

我们在编写代码时既可以选择将<input>元素包裹在<label>中，也可以使用for属性来关联<label>和<input>——两种方式都是正确的。但是使用Bootstrap时还是要遵循它的规则，从而得到所需要的样式。有关如何组织HTML的示例，请参考Bootstrap文档（getbootstrap.com/css/#forms）。

在index.html中表示邮箱地址的<input>后面添加单选按钮标记。

```
...
            <div class="form-group">
              <label for="emailInput">Email</label>
              <input class="form-control" type="email" name="emailAddress"
                id="emailInput" value="" placeholder="dr@who.com">
            </div>
            <div class="radio">
              <label>
                <input type="radio" name="size" value="short">
                Short
              </label>
            </div>
```

```
        <div class="radio">
          <label>
            <input type="radio" name="size" value="tall" checked>
            Tall
          </label>
        </div>
        <div class="radio">
          <label>
            <input type="radio" name="size" value="grande">
            Grande
          </label>
        </div>
      </form>
...
```

将这三个radio的name属性设置为相同的值（size）。这会告诉浏览器，一次只能选择（或“选中”）其中的一个。为**大杯**单选按钮添加一个名为checked的布尔属性，它与autofocus的工作方式一样：当存在时，属性值为true；当不存在时，则为false。

保存index.html，并查看新的单选按钮（如图9-9所示）。

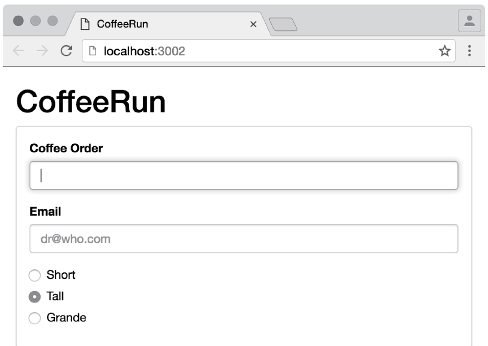


图9-9　咖啡杯型单选按钮

尝试点击单选按钮或者其旁边的文本。无论采用哪种方式，该单选按钮都应该被选中。

9.2.3　添加下拉菜单

有些人对调味咖啡情有独钟。可以给他们提供几种不同的口味以供选择。默认情况下不添加任何口味。

我们当然可以使用一组单选按钮，但是列表中可能有很多种口味。为了确保口味选择部分不会打乱UI，我们将使用下拉菜单。

要创建一个Bootstrap样式的下拉菜单，需要向index.html添加一个类名为form-group的<div>。创建一个类名为form-control的<select>元素，Bootstrap将会把此元素设置为下拉菜单。把<select>和它的<label>关联，<label>的id为flavorShot。在<select>内为每个要显

示的菜单项添加一个<option>元素，给每个菜单项一个相应的值。

```
...
            <div class="radio">
              <label>
                <input type="radio" name="size" value="grande">
                Grande
              </label>
            </div>
            <div class="form-group">
              <label for="flavorShot">Flavor Shot</label>
              <select id="flavorShot" class="form-control" name="flavor">
                <option value="">None</option>
                <option value="caramel">Caramel</option>
                <option value="almond">Almond</option>
                <option value="mocha">Mocha</option>
              </select>
            </div>
          </form>
        </div>
      </div>
...
```

每个<option>元素提供一个可能的值，而<select>元素指定name值。

保存index.html，并检查下拉列表是否正常显示，且下拉框中有你所添加的所有选项（如图9-10所示）。

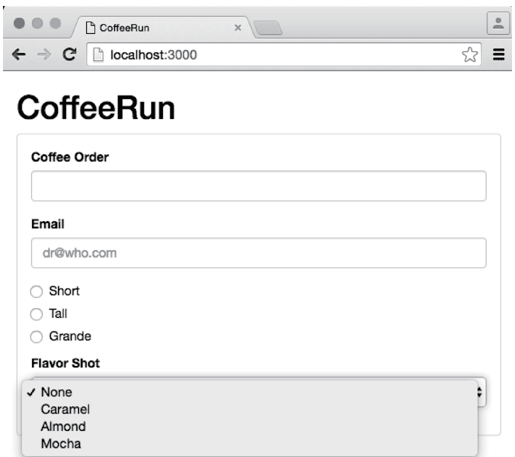


图9-10　咖啡口味下拉菜单

默认情况下是第一个<option>元素被选中。如果想自动选择其他的选项（而不是第一个），可以为将要选择的option元素添加一个selected的布尔属性。

应将第一个下拉项的value属性设置为空字符串。如果完全删除了value属性，浏览器将使用"None"作为value的值。我们永远不应该假设浏览器会按照我们的预想来做，所以这里最好设置value属性。

9.2.4　添加范围滑块

不是每个人都喜欢浓到发苦的咖啡，我们想让用户为他们的咖啡浓度选择一个介于0~100之间的值。另一方面，我们也不希望他们必须手动键入一个确切的值。

为此，可以在index.html中添加一个类型为range的<input>元素，这将创建一个**范围滑块**。<input>和<label>元素应该关联起来，并包含在类名为form-group的<div>中。为了方便客户使用，将默认值设定为30。

```
...
            <option value="mocha">Mocha</option>
          </select>
        </div>
        <div class="form-group">
          <label for="strengthLevel">Caffeine Rating</label>
          <input name="strength" id="strengthLevel" type="range" value="30">
        </div>
      </form>
...
```

保存index.html，并在浏览器中试试这个新滑块，如图9-11所示。

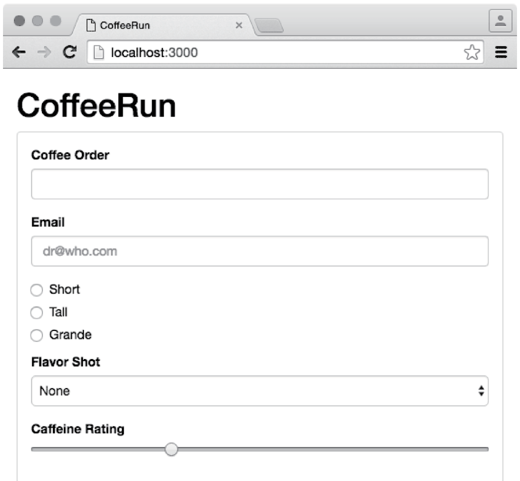


图9-11　咖啡浓度选择滑块

9.2.5　添加提交按钮和重置按钮

最后要添加一个**提交**按钮。同时，以防有的用户想要重新填写，还应该再添加一个**重置**按钮来清除表单。

通常，提交按钮只是一个类型为submit的<input>元素；同样，重置按钮是一个类型为reset的<input>元素。但是为了要利用Bootstrap的CSS，我们将使用<button>元素来替代它们。

在index.html中添加两个类名为`btn btn-default`的`<button>`元素。将第一个的`type`设置为`submit`，将第二个的`type`设置为`reset`。在开始标签和结束标签之间，将`Submit`和`Reset`作为描述文本。

```
...
          <div class="form-group">
            <label for="strengthLevel">Caffeine Rating</label>
            <input name="strength" id="strengthLevel" type="range" value="30">
          </div>
          <button type="submit" class="btn btn-default">Submit</button>
          <button type="reset" class="btn btn-default">Reset</button>
        </form>
...
```

保存更改，浏览器将在表单底部添加按钮（如图9-12所示）。

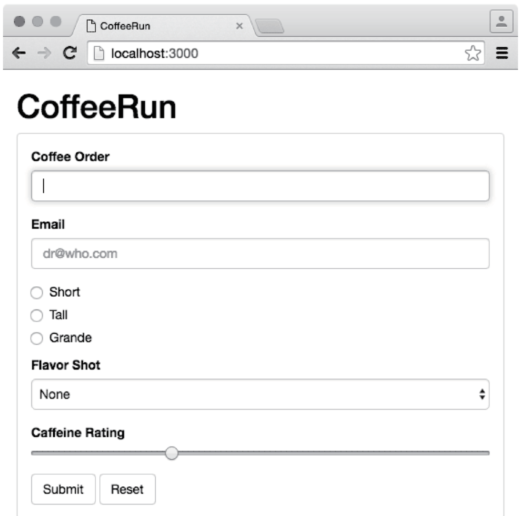


图9-12　提交按钮与重置按钮

提交按钮暂时不会执行任何操作，这部分内容将在下一章讲解。但是**重置**按钮会将表单的值重置为默认值。

这两个按钮有一对类名看似是多余的，其实这纯粹是Bootstrap为了样式而制定的一种约定。`btn`类为Bootstrap按钮提供了所有标准的视觉属性，这其中包括圆角和内边距。`btn-default`类为按钮添加了一个白色的背景。

我们已经使用Bootstrap UI框架设置了CoffeeRun应用程序的样式。通过使用Bootstrap的标记和类名模式，这个应用程序在各种尺寸的屏幕和浏览器版本上都拥有一致的外观和体验。

要了解有关Bootstrap的更多内容，请参阅getbootstrap.com/css上的优秀文档。

Bootstrap特别适合快速为应用程序添加样式，好让我们专注于应用程序逻辑。在接下来的几章中，我们也会这样做。

第 10 章 使用JavaScript处理表单

CoffeeRun的编写工作已经有了很好的开始。到目前为止，它有两个处理其内部逻辑的JavaScript模块和一个Bootstrap样式的HTML表单。在本章中，我们将编写一个更复杂的模块，将表单和逻辑关联，以便于用户使用表单输入咖啡订单。

回顾第2章，浏览器通过使用特定URL发送信息请求与服务器进行通信。具体来说，对于要加载的每个文件，浏览器都要向该文件所在的服务器发送一个`GET`请求。

当浏览器需要**向**服务器发送信息时（比如当用户填写并提交表单时），浏览器会提取表单数据，并将其放入`POST`请求中。服务器接受请求，处理数据，然后返回一个响应（如图10-1所示）。

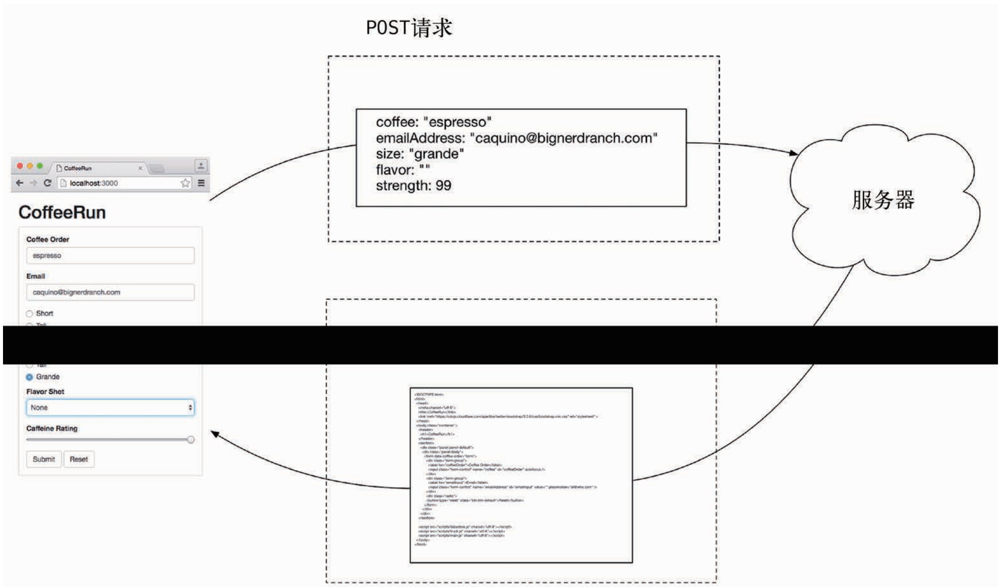


图10-1　传统的服务器端表单处理

在CoffeeRun中，我们不需要将表单数据发送到服务器进行处理。`Truck`和`DataStore`模块起到了与传统服务器端代码相同的作用，它们的工作是为应用程序处理业务逻辑和数据存储。

因为这些代码保存在浏览器中而不是服务器上，所以我们需要在表单提交数据之前捕获其数

据。本章将创建一个新的FormHandler模块来捕获数据。此外，还将把jQuery库添加到CoffeeRun中进行辅助开发。接下来的几章将多次用到jQuery的强大功能来帮助我们开发CoffeeRun。

10.1　创建 FormHandler 模块

FormHandler模块会阻止浏览器向服务器发送表单数据。作为替代方式，它在用户单击**提交**按钮时从表单中读取值,然后使用我们在第8章中编写的createOrder方法将该数据发送到Trunk实例（如图10-2所示）。

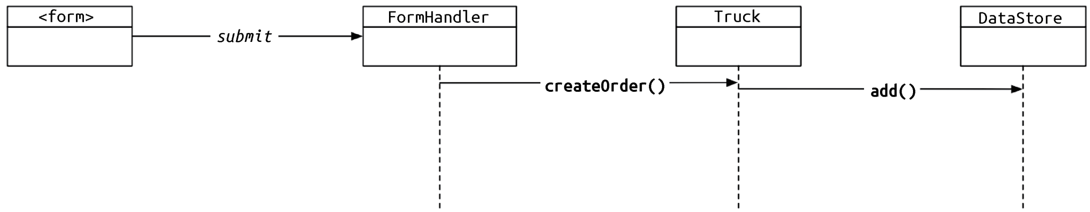


图 10-2　带有 App.FormHandler 的 CoffeeRun 应用程序体系结构

先在scripts文件夹中创建一个名为formhandler.js的新文件，并在index.html中为它添加一个<script>标签。

```
...
          </form>
        </div>
      </div>
    </section>
    <script src="scripts/formhandler.js" charset="utf-8"></script>
    <script src="scripts/datastore.js" charset="utf-8"></script>
    <script src="scripts/truck.js" charset="utf-8"></script>
    <script src="scripts/main.js" charset="utf-8"></script>
  </body>
</html>
```

像其他模块一样,FormHandler将使用IIFE封装代码,并将一个构造函数附加到window.App属性。

打开scripts/formhandler.js并创建一个IIFE，在IIFE中创建一个App变量，把window.App的现有值赋值给它。如果window.App不存在,将一个空对象字面值赋值给它。声明一个FormHandler构造函数，并将其导出到window.App属性。

```
(function (window) {
  'use strict';
  var App = window.App || {};

  function FormHandler() {
    // 放置要运行的代码
  }
```

```
  App.FormHandler = FormHandler;
  window.App = App;

})(window);
```

到目前为止，这段代码依然遵循我们在Trunk和DataStore模块中所使用的模式。但是很快它就会变得不同，因为我们将引入jQuery用于开发。

10.1.1 jQuery简介

jQuery库由John Resig于2006年创建，是最受欢迎的通用开源JavaScript库之一。此外，它为DOM操作、元素创建、服务器通信和事件处理提供了快速便捷的方法。

熟悉jQuery是非常有用的，因为很多代码都使用它来编写。另外，许多库也遵循jQuery的书写和使用约定。事实上，jQuery直接影响了标准的DOM API（document.querySelector和document.querySelectorAll就是受此影响的两个例子）。

然而现在我们并不会深入讲解jQuery，而是依照使用需求来介绍它，从而让它能更好地帮助我们构建CoffeeRun更复杂的部分。如果想进一步探索jQuery，请查阅jquery.com上的文档。

和Bootstrap一样，从cdnjs.com添加一个jQuery的副本到项目中。到cdnjs.com/libraries/jquery找到2.1.4版本并复制其地址。（可能有更高版本，但是在CoffeeRun中应使用2.1.4版本来避免兼容性问题。）

在index.html中添加一个jQuery的<script>标签。

```
...
      </div>
    </section>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.4/jquery.min.js"
      charset="utf-8"></script>
    <script src="scripts/formhandler.js" charset="utf-8"></script>
    <script src="scripts/datastore.js" charset="utf-8"></script>
    <script src="scripts/truck.js" charset="utf-8"></script>
    <script src="scripts/main.js" charset="utf-8"></script>
...
```

保存index.html.

10

10.1.2 导入jQuery

FormHandler将会像导入App一样导入jQuery，这样做是为了指明模块是在使用在别处定义的代码。这是有助于团队成员互相协调和日后维护的最佳做法。

在formhandler.js中创建一个名为$的局部变量，然后赋值为window.jQuery。

```
(function (window) {
  'use strict';
  var App = window.App || {};
  var $ = window.jQuery;
```

```
    function FormHandler() {
      //放置要运行的代码
    }

    App.FormHandler = FormHandler;
    window.App = App;

  })(window);
```

当我们添加jQuery`<script>`标签时，它创建了一个名为`jQuery`的函数，以及一个指向该函数的名为$的变量。大多数开发人员更喜欢在他们的代码中使用`$`。为了保持一致性，应该在导入`window.jQuery`的时候将其分配给本地变量`$`。

为什么`$`被用作变量名？其实JavaScript变量名称可以包含字母、数字、下划线（`_`）或者美元符号（`$`）。（它们只能以字母、下划线或者美元符号**开头**，但不能以数字开头。）而jQuery的作者之所以选择`$`作为变量名称，是因为它简短，而且不太可能被项目中的其他代码所使用。

10.1.3　使用selector参数配置FormHandler实例

`FormHandler`模块应该可以与任何一个`<form>`元素配合使用。为了实现这一点，可以给`FormHandler`构造函数传递一个`selector`，而这个`selector`匹配index.html中`<form>`元素。

更新formhandler.js，为`FormHandler`构造函数添加一个名为`selector`的参数。如果未传入`selector`，则抛出`Error`。

```
(function (window) {
  'use strict';
  var App = window.App || {};
  var $ = window.jQuery;

  function FormHandler(selector) {
    //放置要运行的代码
    if (!selector) {
      throw new Error('No selector provided');
    }
  }

  App.FormHandler = FormHandler;
  window.App = App;

})(window);
```

`Error`是一种内置类型，可以明确地表示代码中出现了意外的值或条件。但目前你使用的`Error`实例只是在控制台打印出消息。

保存并尝试不传递参数来实例化一个新的`FormHandler`对象（如图10-3所示）。（如果browser-sync尚未打开，记得打开。）

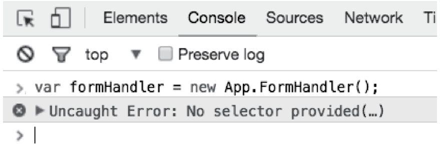


图10-3 不传递参数，实例化`FormHandler`

这是提高`FormHandler`复用性的第一步。在Ottergram中，我们为在DOM代码中用到的选择器创建了变量。而有了`FormHandler`模块后，就无须再那么做了。相反，可以使用传递给构造函数的`selector`参数和jQuery来找到相应的元素。

jQuery常用于在DOM中查找元素。要做到这一点，可以调用jQuery的`$`函数，并将一个字符串作为选择器传递给它。实际上，使用它的方式与使用`document.querySelectorAll`的方式是相同的（尽管jQuery在底层的工作方式不同，这一点稍后会解释）。通常我们称之为使用jQuery“从DOM中选择元素”。

在formhandler.js中声明一个名为`$formElement`的实例变量，然后使用`selector`在DOM中找到匹配的元素，再将结果赋值给`this.$formElement`。

```
(function (window) {
  'use strict';
  var App = window.App || {};
  var $ = window.jQuery;

  function FormHandler(selector) {
    if (!selector) {
      throw new Error('No selector provided');
    }

    this.$formElement = $(selector);
  }

  App.FormHandler = FormHandler;
  window.App = App;

})(window);
```

带`$`前缀的变量表示这个变量是通过jQuery选择出来的元素。虽然使用jQuery时不是必须要使用这个前缀，不过这是许多前端开发人员的常见书写惯例。

当使用jQuery的`$`函数来选择元素时，它并不会像`document.querySelectorAll`一样返回DOM元素的引用，而是返回单个对象，而该**对象**中会包含对所选元素的引用。这个对象同时具有操作引用集合的特殊方法，被称为“jQuery封装集合”。

接下来，要确保元素选择器成功地从DOM检索到了一个元素。如果未查找到任何元素，jQuery将会返回空——如果选择器没有匹配到任何元素，它不会抛出一个异常。因此需要手动检查一下，因为`FormHandler`没有元素是不能工作的。

jQuery封装集合的长度可以告诉我们有多少个匹配元素。更新formhandler.js来检查

this.$formElement的length属性。如果等于0，则抛出Error。

```
(function (window) {
  'use strict';
  var App = window.App || {};
  var $ = window.jQuery;

  function FormHandler(selector) {
    if (!selector) {
      throw new Error('No selector provided');
    }

    this.$formElement = $(selector);
    if (this.$formElement.length === 0) {
      throw new Error('Could not find element with selector: ' + selector);
    }
  }

  App.FormHandler = FormHandler;
  window.App = App;

})(window);
```

FormHandler构造函数会根据传入的selector参数来处理<form>元素。此外，它还使用实例变量保留了对<form>元素的引用，这保证了代码不会对DOM进行无谓的遍历，这是一个最佳性能实践。（另一种方法是一次次地调用$来重新选择相同的元素。）

10.2　添加提交处理程序

下一步是让FormHandler监听<form>元素上的submit事件，从而在提交时执行回调函数。

为了提高FormHandler模块的复用性，不要对提交处理程序进行硬编码，而是写一个接受函数参数的方法，添加一个提交监听器，然后在监听器的内部调用这个函数参数。

首先，在formhandler.js添加一个名为addSubmitHandler的原型方法。

```
...
    if (this.$formElement.length === 0) {
      throw new Error('Could not find element with selector: ' + selector);
    }
  }

  FormHandler.prototype.addSubmitHandler = function () {
    console.log('Setting submit handler for form');
    // 放置要运行的代码
  };

App.FormHandler = FormHandler;
...
```

我们将使用jQuery的on方法来替代在Ottergram中使用的addEventListener方法。它和类addEventListener类似，但提供了更大的便利性。现在，通过使用addEventListener的方式

来使用它。（下一章将用到on提供的便利性。）

```
...
    if (this.$formElement.length === 0) {
      throw new Error('Could not find element with selector: ' + selector);
    }
  }

  FormHandler.prototype.addSubmitHandler = function () {
    console.log('Setting submit handler for form');
    // 放置要运行的代码
    this.$formElement.on('submit', function (event) {
      event.preventDefault();
    });
  };
...
```

on方法接受一个事件名称，并在事件被触发时执行回调。回调函数应该接受该事件的事件对象。调用event.preventDefault是为了确保用户提交表单时不会离开CoffeeRun页面。（我们对Ottergram中的缩略图链接做了同样的处理。）

10.2.1 提取数据

提交表单时，代码应该从表单中读取用户的输入，然后对其进行**处理**。在formhandler.js的提交处理程序中，创建一个名为data的新变量，并为它赋值一个对象字面量，确保它可以保存表单中每个元素的值。

```
...
  FormHandler.prototype.addSubmitHandler = function () {
    console.log('Setting submit handler for form');
    this.$formElement.on('submit', function (event) {
      event.preventDefault();

    var data = $(this).serializeArray();
    console.log(data);
    });
  };
...
```

在提交处理程序的回调中，this对象是对form元素的引用。jQuery提供了一个便捷的方法（serializeArray）来从表单获取值。为了使用serializeArray，需要使用jQuery包装表单。调用$(this)会返回一个包装对象，这个包装对象可以使用serializeArray方法。

serializeArray以对象数组的形式返回表单数据。我们将它赋值给一个名为data的临时变量，并输出到控制台。要了解serializeArray的输出情况，请保存文件并在控制台运行以下代码：

```
var fh = new App.FormHandler('[data-coffee-order="form"]');
fh.addSubmitHandler();
```

接下来在表单中填写一些测试数据，然后单击提交按钮，就可以看到打印到控制台的数组。

单击对象数组旁边的▶，会看到如图10-4所示的内容。

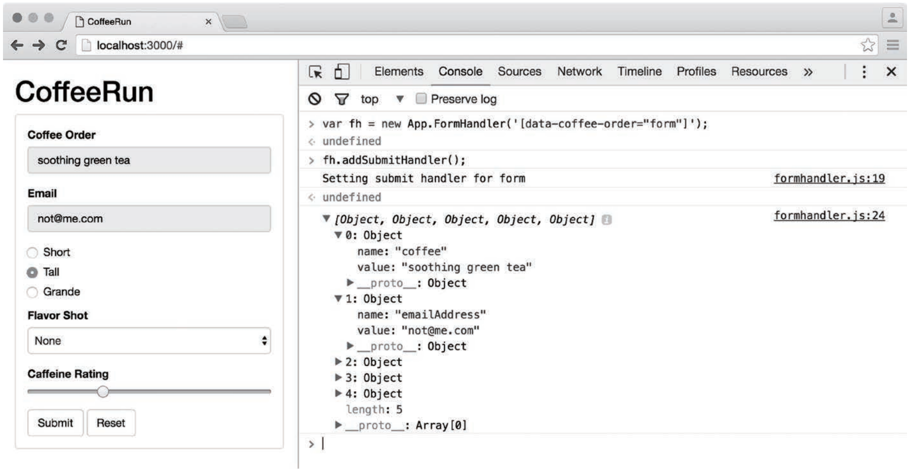


图10-4　serializeArray将表单数据作为对象数组返回

可以看到数组中每个对象都有一个对应于<form>元素的name属性的键名，以及用户为该元素提供的键值。

现在可以遍历数组并复制每个元素的值。在formhandler.js中为serializeArray添加一个forEach方法，并给forEach传递一个回调方法。它会对数组中的每个对象执行回调并使用相应对象的name和value在data上创建一个新属性。

```
...
  FormHandler.prototype.addSubmitHandler = function () {
    console.log('Setting submit handler for form');
    this.$formElement.on('submit', function (event) {
      event.preventDefault();

      var data = $(this).serializeArray(); {};
      $(this).serializeArray().forEach(function (item) {
        data[item.name] = item.value;
        console.log(item.name + ' is ' + item.value);
      });
      console.log(data);

    });
  };
...
```

要查看实际效果，请保存修改后的代码，并在填写表单之前，在控制台上再次运行测试代码：

```
var fh = new App.FormHandler('[data-coffee-order="form"]');
fh.addSubmitHandler();
```

当填写完表单并单击提交按钮后，应该能看到输入的信息被复制到data对象，并被输出到控制台上（如图10-5所示）。

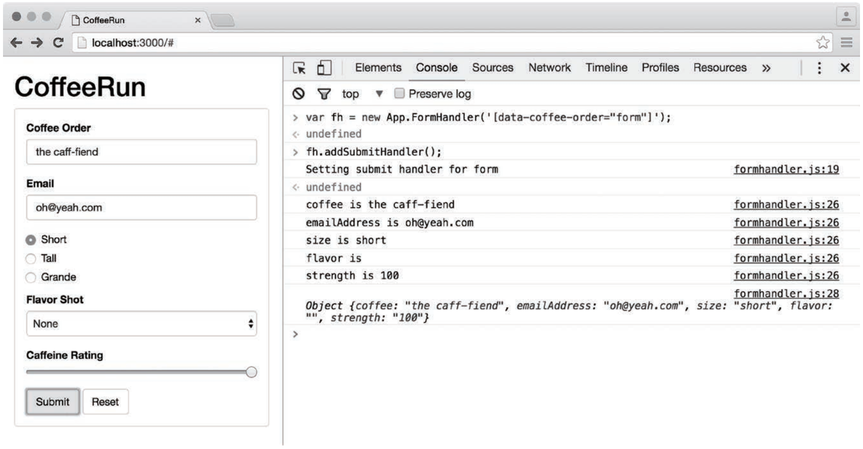


图10-5　表单数据在迭代的回调中被复制

10.2.2　接受并调用回调函数

既然我们已经将表单数据保存为了一个对象，接下来需要将这个对象传递到Truck实例的createOrder方法中。但是FormHandler无法访问Truck实例。（在这里创建一个新的Truck实例也并不明智。）

可以给addSubmitHandler传递一个函数参数来解决这个问题，这个函数可以在事件处理程序中被调用。

在formhandler.js中添加一个名为fn的参数。

```
...
  FormHandler.prototype.addSubmitHandler = function (fn) {
    console.log('Setting submit handler for form');
    this.$formElement.on('submit', function (event) {
      event.preventDefault();
...
```

每当表单的submit事件在浏览器中被触发时，submit事件处理程序回调就会被执行。此时，我们期望调用fn函数。

在formhandler.js的提交处理程序回调中调用fn，并把包含用户输入的数据对象传递给fn。

```
...
  FormHandler.prototype.addSubmitHandler = function (fn) {
    ...
      console.log(data);
      fn(data);
    });
  };
...
```

当一个FormHandler实例被创建后，就可以给addSubmitHandler传递任何回调了。于是只

要表单被提交，回调函数就会被调用，并会将用户在表单中输入的数据传递给该回调函数。

10.3　使用 FormHandler

我们需要在main.js中实例化一个FormHandler实例，并把<form>元素的选择器[data-coffee-order="form"]传递给它。在main.js的顶部为此选择器创建一个变量，以便在需要时可以重复使用。

```
(function (window) {
  'use strict';
  var FORM_SELECTOR = '[data-coffee-order="form"]';
  var App = window.App;
...
```

接下来创建一个名为FormHandler的局部变量并为其赋值App.FormHandler。

```
(function (window) {
  'use strict';
  var FORM_SELECTOR = '[data-coffee-order="form"]';
  var App = window.App;
  var Truck = App.Truck;
  var DataStore = App.DataStore;
  var FormHandler = App.FormHandler;
  var myTruck = new Truck('ncc-1701', new DataStore());
...
```

在main.js模块的末尾使用FORM_SELECTOR作为参数调用FormHandler构造函数，这样可以确保FormHandler实例与选择器选出的DOM元素绑定在一起。再将这个实例赋值给一个名为FormHandler的新变量。

```
...
  var Truck = App.Truck;
  var DataStore = App.DataStore;
  var FormHandler = App.FormHandler;
  var myTruck = new Truck('ncc-1701', new DataStore());
  window.myTruck = myTruck;
  var formHandler = new FormHandler(FORM_SELECTOR);

  formHandler.addSubmitHandler();
  console.log(formHandler);
})(window);
```

保存代码并返回浏览器，控制台应该输出了Setting submit handler for form，这表示在页面加载完成之后addSubmitHandler被调用了。但如果我们现在填写并提交表单，它会报错（如图10-6所示）。

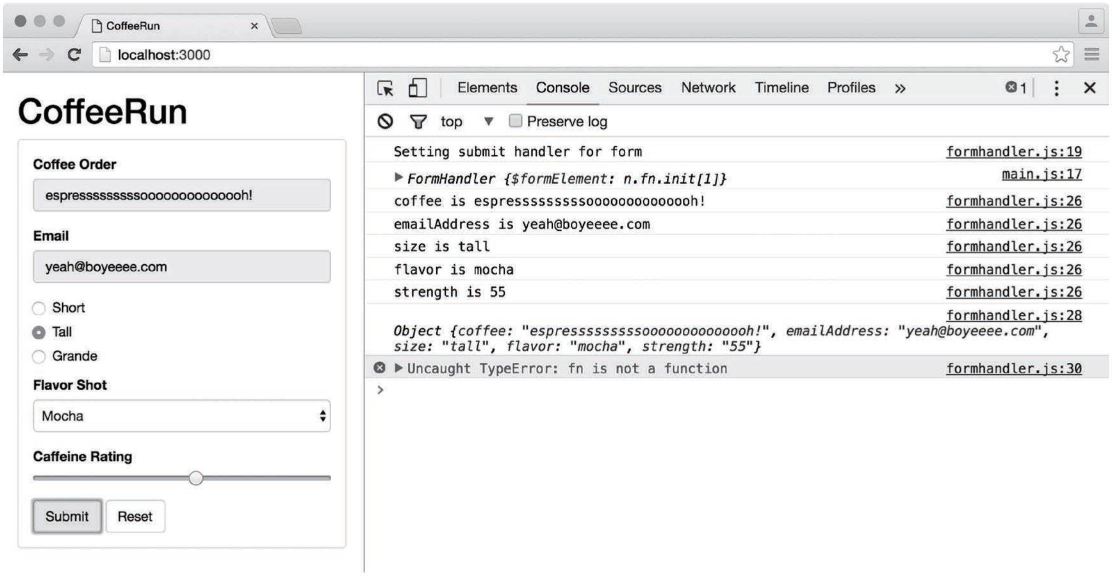


图10-6　在页面加载后调用addSubmitHandler

这是因为我们没有向addSubmitHandler传递任何内容，下一节将纠正这个错误。

将createOrder注册为提交处理程序

我们希望每次发生submit事件时都调用createOrder，但是我们不能只传递一个createOrder的引用到formHandler.addSubmitHandler，因为 createOrder在事件处理回调中被调用时，它的**所有者**会有所变化。此时，createOrder内部的this将不再是Trunk实例，这导致了createOrder运行时报错。

应该把myTruck.createOrder的所有者绑定为myTruck，然后再把这个函数传给formHandler.addSubmitHandler。

保存修改后的main.js，确保已经对方法进行了绑定以保证其所有者为myTruck。

```
...
  window.myTruck = myTruck;
  var formHandler = new FormHandler(FORM_SELECTOR);

  formHandler.addSubmitHandler(myTruck.createOrder.bind(myTruck));
  console.log(formHandler);
})(window);
```

我们能对原始原型方法的定义使用bind么？定义原型方法时，只能在方法**内部**访问实例。bind要求我们提供被调用函数的预期所有者的引用——一个必须能在方法体外使用的引用。由于我们无法在方法体外获取引用实例，所以无法对原始原型方法使用bind。

保存修改并填写表单。提交表单后，myTruck.printOrders方法应该可以被调用，填写的

数据也就可以被添加到如图10-7所示的订单渲染列表中。

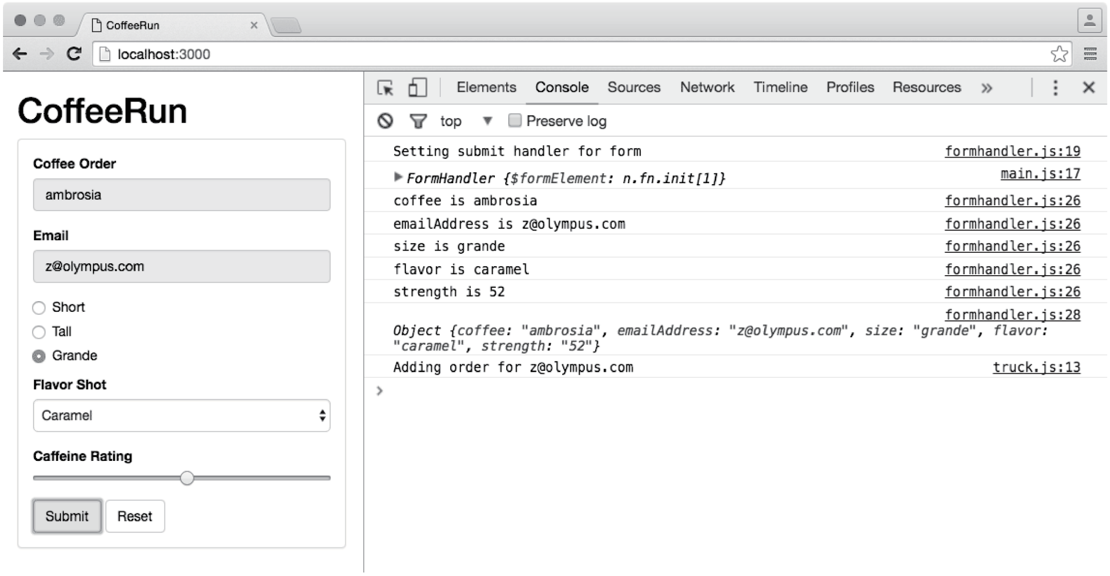


图 10-7　提交表单时 createOrder 被调用

10.4　UI 优化

如果提交表单之后，旧数据能够被清除，那么用户便可以立即输入下一个订单——这将是一个不错的优化。重置表单就像调用`<form>`的`reset`方法一样简单。

在formhandler.js中找到`FormHandler.prototype.addSubmitHandler`方法。在`this.$formElement.on('submit'…)`回调函数的末尾调用表单的`reset`方法。

```
...
  FormHandler.prototype.addSubmitHandler = function (fn) {
    console.log('Setting submit handler for form');
    this.$formElement.on('submit', function (event) {
      event.preventDefault();

      var data = {};
      $(this).serializeArray().forEach(function (item) {
        data[item.name] = item.value;
        console.log(item.name + ' is ' + item.value);
      });
      console.log(data);
      fn(data);
      this.reset();
    });
  };
...
```

保存修改，并在表单中填写一些数据。提交表单时，数据已经被清除。

最后再为UI做一个调整。就像在上一章中所看到的，当表单可以输入数据的时候，它会被聚焦。为了让表单的特定字段被聚焦，可以调用它的focus方法。（给咖啡订单添加的autofocus属性只会在网页首次加载时有效。）

可以通过表单的elements属性轻松获得各个表单字段。elements是表单字段数组，可以从0开始索引并引用。

在formhandler.js的提交处理程序回调中调用this.reset之后，在表单第一个字段上调用focus方法。

```
...
  FormHandler.prototype.addSubmitHandler = function (fn) {
    console.log('Setting submit handler for form');
    this.$formElement.on('submit', function (event) {
      event.preventDefault();

      var data = {};
      $(this).serializeArray().forEach(function (item) {
        data[item.name] = item.value;
        console.log(item.name + ' is ' + item.value);
      });
      console.log(data);
      fn(data);
      this.reset();
      this.elements[0].focus();
    });
  };
...
```

CoffeeRun现在得到了jQuery的强力支持并且可以接受用户输入了！这样HTML和JavaScript模块之间的联系便建立了起来。在下一章中，我们将基于从表单中获取的数据来创建交互式DOM元素，从而完成蓝图。

10.5　初级挑战：添加超级尺寸

为咖啡订单添加另一个杯型选项——一个响亮的名字，例如“哥斯拉咖啡”。

用特大杯型来添加一个新订单，并在控制台检查应用程序数据，确保数据能够正确保存。

10.6　中级挑战：当滑块滑动时显示其数值

为滑块的change事件创建处理程序。当滑块滑动时，在label标签旁边显示相应的数字。

作为额外挑战，要通过改变数字（或者label标签）的颜色来反应咖啡的浓度——用绿色表示淡咖啡，用黄色表示常规浓度的咖啡，用红色表示非常浓的咖啡。

10.7　高级挑战：添加选择

当用户提交一个最大杯、最高浓度的调味咖啡订单时，他会解锁成就——弹出一个Bootstrap模态框让他确认选择的浓度和口味。此外，还会询问他是否坚持自己的选择。如果是的话，在邮箱字段已经输入内容的情况下，再添加一个额外的表单字段。这个字段允许他定制自己想要的咖啡，例如打发时间型、心灵阅读型或者解决问题型咖啡。

有关如何包含和触发Bootstrap的模态行为，请参阅getbootstrap.com/javascript上的文档。（需要添加一个`<script>`标签来引用cdnjs.com上Bootstrap的JavaScript。）

第 11 章

从数据到DOM

在上一章中，我们构建了FormHandler模块，它在与用户交互的表单和其他代码之间搭建了沟通的桥梁。通过拦截表单的提交事件，将用户的输入提交到Trunk模块，Trunk模块再把数据保存到DataStore实例。

这一章将构建另一段UI代码——CheckList模块。和Truck模块一样，它也会从FormHandler模块中接受数据，但它主要专注于将待处理的订单添加到页面的清单上。当这个清单中的任意一项被点击时，CheckList模块就把它从页面中移除，同时通知Truck模块将该订单从DataStore中移除。图11-1显示了配备有待处理订单清单的CoffeeRun。



图11-1　让订单来得更多一些吧

11.1　建立清单

我们将继续使用Bootstrap类来设置表单元素的样式。从index.html开始，像制作咖啡订单表单时那样，添加类名为`panel panel-default`的`<div>`，并在其内部定义一个类名为`panel-body`的`<div>`。再在它们内部添加一个标题和另一个用于展示清单项的`<div>`。而这整个类名为`panel panel-default`的`<div>`标记，应该添加在表单之后。

```
...
    <header>
      <h1>CoffeeRun</h1>
    </header>
    <section>
      <div class="panel panel-default">
        <div class="panel-body">
          <form data-coffee-order="form">
            ...
          </form>
        </div>
      </div>

      <div class="panel panel-default">
        <div class="panel-body">
          <h4>Pending Orders:</h4>
          <div data-coffee-order="checklist">
          </div>
        </div>
      </div>
    </section>
...
```

和之前一样，使用`<div>`标签来展现Bootstrap的样式。清单的主体部分是`[data-coffee-order="checklist"]`元素。在Javascript创建单个咖啡订单后，它将作为在DOM上展示清单项的目标元素。

保存index.html，启动 browser-sync，检查一下CoffeeRun是否显示了一个空的**待处理订单清单**区域（如图11-2所示）。

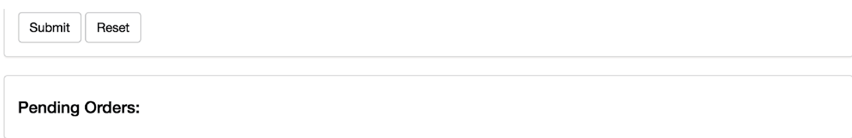


图11-2　添加清单项标记之后

现在可以回归JavaScript了。

11.2　创建 CheckList 模块

在scripts文件夹中创建一个名为checklist.js的新文件，并在index.html中添加一个`<script>`链接。

```
...
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.4/jquery.js"
      charset="utf-8"></script>
    <script src="scripts/checklist.js" charset="utf-8"></script>
    <script src="scripts/formhandler.js" charset="utf-8"></script>
    <script src="scripts/datastore.js" charset="utf-8"></script>
    <script src="scripts/truck.js" charset="utf-8"></script>
    <script src="scripts/main.js" charset="utf-8"></script>
...
```

首先保存index.html，在checklist.js中添加标准的IIFE模块代码；然后导入App命名空间和jQuery，并将它们赋值给一个局部变量；接下来为CheckList创建一个构造函数，记住要为构造函数传递一个selector参数，并且这个selector参数要至少能够匹配DOM中的一个元素。在IIFE的最后，将CheckList构造函数导出为App命名空间的一部分。

```
(function (window) {
  'use strict';

  var App = window.App || {};
  var $ = window.jQuery;

  function CheckList(selector) {
    if (!selector) {
      throw new Error('No selector provided');
    }

    this.$element = $(selector);
    if (this.$element.length === 0) {
      throw new Error('Could not find element with selector: ' + selector);
    }
  }

  App.CheckList = CheckList;
  window.App = App;
})(window);
```

CheckList模块需要3个方法来完成其工作：第1个负责创建一个清单项，这个清单项包括了复选框和描述文本。可以将清单项视为table中的一行。第2个方法会从table中移除一行。第3个方法会为单击事件添加一个监听器，从而让代码知道**何时**需要移除一行。

现在要处理的第1个方法会为新订单创建新的一行。图11-3展示了CheckList如何在提交表单时将清单项添加到页面上。

11

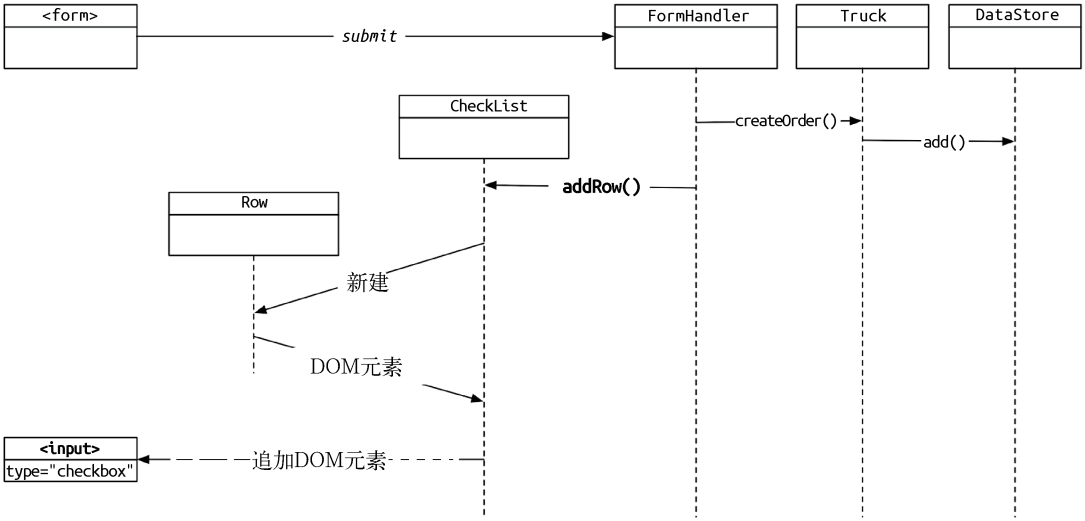


图11-3　提交订单表单时的订单事件

11.3　创建行构造函数

我们不能在index.html中直接为清单项创建标记，因为为了响应表单提交事件，它们需要在页面渲染之后再被添加。所以，要在CheckList模块添加一个Row构造函数。

Row构造函数将负责创建所有用于表示单个咖啡订单的DOM元素，包括复选框和描述文本。但是Row构造函数并不会被导出到App命名空间，它只会被CheckList.prototype内部的方法使用。

在checklist.js中的App.CheckList = CheckList;之前添加Row构造函数，它应该接受一个名为coffeeOrder的参数。coffeeOrder与传递给Truck.prototype.createOrder的数据是相同的。

```
...
    this.$element = $(selector);
    if (this.$element.length === 0) {
      throw new Error('Could not find element with selector: ' + selector);
    }
  }

  function Row(coffeeOrder) {
     // 放置要运行的构造函数代码
  }

  App.CheckList = CheckList;
  window.App = App;
})(window);
```

使用jQuery创建DOM元素

Row构造函数会使用jQuery来构建DOM元素。我们应该为组成清单项的各个元素声明变量。然后，如图11-4所示，构造函数会将它们一起添加到DOM元素的**子树**中。CheckList将会获取该子树，并将其作为[data-coffee-order="checklist"]元素的子元素附加到页面的DOM树上。

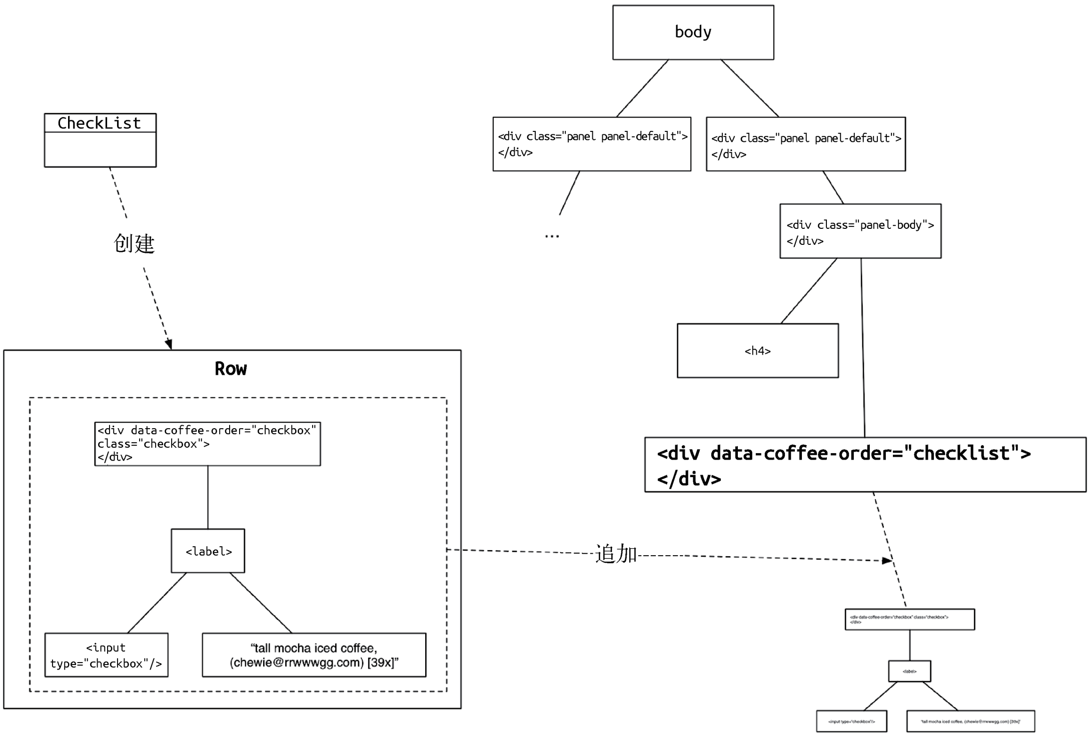


图11-4 CheckList创建一行并将其添加到DOM元素上

（订单描述信息中的“[39x]”表示咖啡因的浓度。）

由图11-4的Row构造函数所创建的DOM子树等同于以下标记：

```
<div data-coffee-order="checkbox" class="checkbox">
  <label>
    <input type="checkbox" value="chewie@rrwwwgg.com">
    tall mocha iced coffee, (chewie@rrwwwgg.com) [39x]
  </label>
</div>
```

一个带有类名checkbox的<div>会被用于容纳<label>和<input>元素。checkbox类会让<div>采用适当的Bootstrap样式。当我们在复选框上触发单击操作时，Javascript代码会使用data-coffee-order属性。

注意，<input>的type属性也需要是checkbox，这让浏览器把输入框绘制为复选框表单元

素。订单的纯文本描述紧跟在<input>之后。<label>元素包含了复选框和纯文本描述，这将使文本和输入框都成为复选框的可点击范围。

我们每次都会创建<label>、<div>和<input>元素，然后手动将其嵌套以构建一个DOM子树。稍后我们会将这个DOM子树附加到活动DOM（页面上当前显示的DOM）上。另外，还要创建一个用于保存订单的描述文本的字符串，例如“tall mocha iced coffee, (chewie@rrwwwgg.com) [39x]”。

为了创建这些元素，使用jQuery的$函数。到目前为止，我们只使用$函数从DOM中选择过元素，其实它也可以用来创建元素。

首先，在checklist.js里的Row构造函数中调用$函数来创建一个<div>元素，向它传递两个用于描述将要创建的DOM元素的参数。第一个参数为DOM元素的HTML标签，本例中则是'<div></div>'。第二个参数为jQuery应该添加到<div>上的属性对象，对象的键值对会被转化为新元素的属性。

它返回的结果是一个由jQuery创建的DOM元素，我们将它赋值给一个名为$div的新变量。这并不是一个实例变量（也就是说，它只是$div而不是this.$div），$前缀表示它不是一个纯DOM元素，而是jQuery创建的一个引用。

在checklist.js中这样做：

```
...
  function Row(coffeeOrder) {
    // 放置要运行的构造函数代码
    var $div = $('<div></div>', {
      'data-coffee-order': 'checkbox',
      'class': 'checkbox'
    });
  }
...
```

请注意，两个属性名称要使用单引号。你也许认为在使用jQuery创建DOM元素时，应该始终对属性名使用单引号。然而事实并非如此。具有特殊字符（如破折号）的属性名称需要用引号包裹，否则会被视为语法错误。而使用字符表中的字母、数字、下划线（_）和美元符号（$）作为属性名称（或者变量名称）的有效字符时，可以不使用单引号。

'class'在单引号中是因为class是一个JavaScript保留字，因此需要使用单引号来防止浏览器以JavaScript形式对其进行解析（如果不使用单引号也会导致语法错误）。

接下来，在checklist.js中使用$函数创建<label>元素，但是不使用对象参数，因为它不需要额外的属性。

```
...
  function Row(coffeeOrder) {
    var $div = $('<div></div>', {
      'data-coffee-order': 'checkbox',
      'class': 'checkbox'
    });
```

```
    var $label = $('<label></label>');
  }
...
```

现在，调用$函数并为其传递一个`<input>`HTML标签，从而为复选框创建一个`<input>`元素。将第二个参数的`type`指定为`checkbox`，`value`指定为客户的邮箱地址。因为这些属性名称都没有使用特殊字符，所以不必将它们放在单引号之中。

```
...
  function Row(coffeeOrder) {
    var $div = $('<div></div>', {
      'data-coffee-order': 'checkbox',
      'class': 'checkbox'
    });

    var $label = $('<label></label>');

    var $checkbox = $('<input></input>', {
      type: 'checkbox',
      value: coffeeOrder.emailAddress
    });
  }
...
```

通过将`value`的值设置为客户的邮箱地址，可以把复选框和客户的咖啡订单关联起来。在稍后添加单击处理程序时，可以根据`value`属性中的邮箱地址来判断哪个咖啡订单被点击过。

最后要创建的是在复选框旁边显示的描述文本，你将使用+=运算符连接信息片段为它构成一个字符串。

在checklist.js中创建一个名为`description`的变量，并将其设置为订单的`size`属性。接着，添加逗号和空格。如果提供了口味（`flavor`），就使用+=连接它们。然后再连接咖啡(`coffee`)、邮箱地址（`emailAddress`）和浓度（`strength`）值。`emailAddress`应该使用小括号包起来，而浓度（`strength`）则应该使用中括号包起来，且后面跟着字母*x*。（添加小括号和中括号不是因为语法需要，只是为了格式化文本。）

```
...
  function Row(coffeeOrder) {
    ...

    var $checkbox = $('<input></input>', {
      type: 'checkbox',
      value: coffeeOrder.emailAddress
    });

    var description = coffeeOrder.size + ' ';
    if (coffeeOrder.flavor) {
      description += coffeeOrder.flavor + ' ';
    }

    description += coffeeOrder.coffee + ', ';
    description += ' (' + coffeeOrder.emailAddress + ')';
```

```
    description += ' [' + coffeeOrder.strength + 'x]';
  }
...
```

+=连接运算符在一个步骤中同时执行了拼接和赋值两个操作，这意味着以下两行代码是等价的：

```
description += coffeeOrder.flavor + ' ';
description = description + coffeeOrder.flavor + ' ';
```

有了清单项的所有组成部分后，就可以将它们嵌套了。（如图11-5所示）

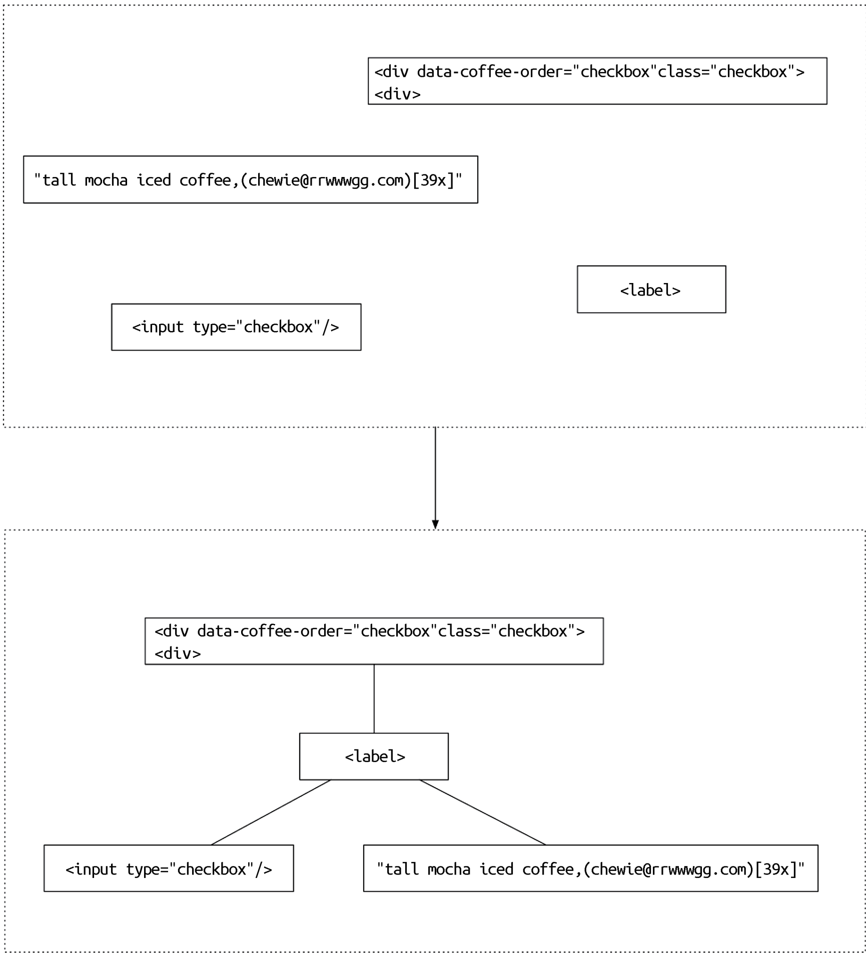


图 11-5　将各个 DOM 元素组合到子树中

分3步来完成此操作。

(1) 把`$checkbox`追加到`$label`中。

(2) 把description追加到$label中。

(3) 把$label追加到$div中。

一般来说，会按照从左到右，从下到上的顺序来构建子树。这种方法和我们在第3章中为Ottergram书写CSS时的规则类似——从最小的元素开始，从内向外的书写。

在checklist.js中使用jQuery的append方法将元素连接在一起。此方法接受DOM元素或者jQuery封装集合作为参数，并将其添加为调用者的子元素。

```
...
  function Row(coffeeOrder) {
    ...
    description += coffeeOrder.coffee + ', ';
    description += ' (' + coffeeOrder.emailAddress + ')';
    description += ' [' + coffeeOrder.strength + 'x]';

    $label.append($checkbox);
    $label.append(description);
    $div.append($label);
  }
...
```

现在的Row构造函数已经可以使用传入的咖啡订单数据创建并组装元素的子树了。然而，因为Row将被当作构造函数而不是普通函数，所以它不能简单地返回这个子树。（实际上，构造函数**永远**不应该有return语句。当对构造函数使用new时，JavaScript会自动返回一个值。）

但是可以在checklist.js中把子树赋值给this.$element，并将其当作实例的属性来访问。（选择此名称只是为了遵循其他构造函数的约定，它本身没有任何特殊的含义。）

```
...
  function Row(coffeeOrder) {
    ...
    $label.append($checkbox);
    $label.append(description);
    $div.append($label);

    this.$element = $div;
   }
...
```

Row构造函数现已准备就绪，可以使用它构建带复选框的DOM子树来表示每个咖啡订单。这个DOM子树被保存在了一个实例变量中。

11.4 在提交时创建清单行

接下来向CheckList添加一个方法，该方法使用Row构造函数创建Row实例。它会将每个Row实例的$element添加到页面的活动DOM上。

在checklist.js中给CheckList.prototype添加一个addRow方法，此方法接受coffeeOrder参数，它是一个包含单个咖啡订单所有数据的对象。

在这个新方法中，调用Row构造函数，并向其传入coffeeOrder参数，从而创建一个新的Row实例；再把新创建的实例赋值给变量rowElement；然后，将rowElement的$element属性（它包含了DOM子树）附加到CheckList实例的$element属性（它是对清单项容器的引用）上。

```
...
    function CheckList(selector) {
      ...
    }

    CheckList.prototype.addRow = function (coffeeOrder) {
      // 使用咖啡订单信息创建一个新的Row实例
      var rowElement = new Row(coffeeOrder);

    // 把新的Row实例的$element属性添加到清单中
    this.$element.append(rowElement.$element);
    };

    function Row(coffeeOrder) {
      ...
```

以上就是将Row的DOM子树添加到页面上所要做的所有工作。保存checklist.js。

在main.js中将匹配整个清单区域的[data-coffee-order="checklist"]选择器赋值到一个变量上。然后，把CheckList模块从APP命名空间导入到本地变量CHECKLIST_SELECTOR中。

```
(function (window) {
  'use strict';
  var FORM_SELECTOR = '[data-coffee-order="form"]';
  var CHECKLIST_SELECTOR = '[data-coffee-order="checklist"]';
  var App = window.App;
  var Truck = App.Truck;
  var DataStore = App.DataStore;
  var FormHandler = App.FormHandler;
  var CheckList = App.CheckList;
  var myTruck = new Truck('ncc-1701', new DataStore());
...
```

现在就可以实例化一个CheckList实例来添加咖啡清单了。

你可能会试图添加另一个对formHandler.addSubmitHandler的调用，但是结果不会如你所愿。为什么呢？这是因为每次调用addSubmitHandler时，它都会注册一个新的回调函数并重置表单（通过调用this.reset）。

想想下面的代码：

```
...
// 实例化一个新的CheckList类型
var checkList = new CheckList(CHECKLIST_SELECTOR);

var formHandler = new FormHandler(FORM_SELECTOR);
formHandler.addSubmitHandler(myTruck.createOrder.bind(myTruck));

// 这并不会按照你所预想的执行
formHandler.addSubmitHandler(checkList.addRow.bind(checkList));
...
```

这段代码注册了两个回调函数，它们会在表单提交时执行。在第一个提交处理程序（myTruck.createOrder）被调用后，表单被重置。在第二个提交处理程序（checkList.addRow）被调用后，表单中并没有留下任何信息。这样会导致虽然数据被添加到了DataStore，但是没有任何一个清单项被添加到页面上。

为了解决这个问题，需要单独传递一个匿名函数到formHandler.addSubmitHandler中，而该匿名函数会调用myTruck.createOrder和checkList.addRow。

此外，这些方法都需要绑定到特定的实例上（这意味着需要设置this值）。虽然我们已经学会了如何使用bind来设置this，但是这里会采用别的方法。

使用call绑定this

使用call与使用bind设置this值的方法类似。两者的区别是，bind返回一个新版本的函数或者方法，但并不会立即执行它；而call实际上会调用返回的函数或者方法，并允许将this设置为传入的第一个参数。（如果需要将其他的参数也传递到函数中，只需要将额外参数添加到参数列表中即可。）call会运行函数体，并返回函数的返回值。

这里需要使用的是call而不是bind，因为除了设置this值之外，还要调用myTruck.createOrder和checkList.addRow。

在main.js中删除formHandler.addSubmitHandler的现有调用，添加formHandler.addSubmitHandler的新调用方式，并传入一个匿名函数。这个匿名函数具有一个data形参。在匿名函数中使用call方法设置myTruck.createOrder和checkList.addRow的this值，同时将data作为第二个参数传递。

```
...
  var myTruck = new Truck('ncc-1701', new DataStore());
  window.myTruck = myTruck;
  var checkList = new CheckList(CHECKLIST_SELECTOR);
  var formHandler = new FormHandler(FORM_SELECTOR);

  formHandler.addSubmitHandler(myTruck.createOrder.bind(myTruck));
                               function (data) {
    console.log(formHandler);
    myTruck.createOrder.call(myTruck, data);
    checkList.addRow.call(checkList, data);
  });
})(window);
```

现在已经有了一个会调用createOrder和addRow的提交处理程序。当这两个函数被调用时，提交处理程序为它们传递了正确的this值和表单中的数据。

保存更改，然后在表单中输入一些数据并提交表单，以此验证清单功能的正确性。每当订单被提交时，都可以看到订单被添加到如图11-6所示的待处理清单中。

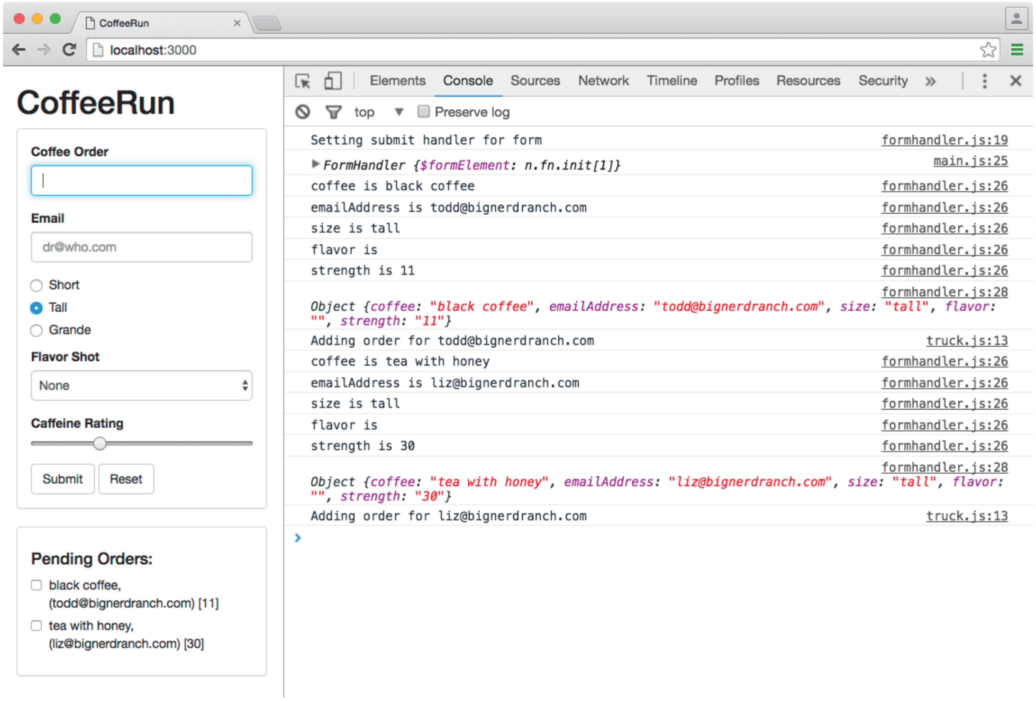


图11-6　提交表单，将订单添加到清单中

11.5　通过单击行完成订单

我们就要完成了！CoffeeRun的用户现在可以填写表单来添加订单。当他们提交表单时，订单信息会被保存到应用程序的数据库中，还会被绘制成清单项。

接下来，用户可以勾掉清单项了。当他们单击清单项时，则意味着订单已完成。这时应该从应用程序数据库中删除订单信息，并把该清单项从页面中删除。图11-7展示了这个过程。

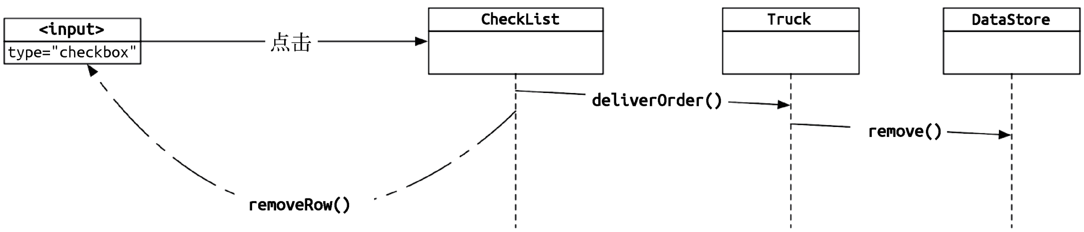


图11-7　单击清单项的顺序图

首先，要创建从页面中删除清单项的功能。

11.5.1　创建CheckList.prototype.removeRow方法

创建Row时，<input>的value被设置成了客户的邮箱地址。removeRow方法将使用邮箱地址参数在UI中检索并移除正确的CheckList项。removeRow将创建一个属性选择器来找到<input>元素，这个元素的value与邮箱地址相匹配。

在找到匹配的元素之后，我们将在DOM上往上查找，直到找到[data-coffee-order="checkbox"]的元素。

而这就是包含清单中一行所有元素的<div>元素。最后，因为使用了jQuery来选择这个<div>，所以可以调用它的remove方法，从DOM中移除元素，并清除所有添加到该DOM子树中任何元素上的事件监听器。

在checklist.js中添加removeRow方法，并指定一个emailAddress参数。使用实例的$element属性来检索它的所有后代元素，其中后代元素的value属性与邮箱参数要相匹配。接着，对检索到的元素调用closest方法，检索该元素的data-coffee-order属性为"checkbox"的祖先。最后，对该祖先执行remove方法。（代码中有一些新语法，之后将对其进行解释。）

```
...
  CheckList.prototype.addRow = function (coffeeOrder) {
    ...
  };

  CheckList.prototype.removeRow = function (email) {
    this.$element
      .find('[value="' + email + '"]')
      .closest('[data-coffee-order="checkbox"]')
      .remove();
};

function Row(coffeeOrder) {
  ...
```

我们在这里**链式调用**了几个方法。jQuery的设计允许我们像多步执行一样对一个对象进行多个方法调用，只需要在最后一个方法调用的末尾加上分号即可。

链式调用的要求是前一个方法必须返回jQuery封装对象，以便调用后一个方法。find返回了一个jQuery封装对象，closest也是如此。这样就可以将三个方法链式调用了。

请注意，this.$element.find只是在一个**范围**内查找，而不是搜索整个DOM——它只会搜索this.$element的后代元素。

11.5.2　删除被覆盖的条目

保存文件，切换到浏览器。在表单中填写两个拥有相同邮箱地址的订单，分别下成“订单1”和“订单2”。在提交两个订单后，在控制台中调用myTruck.printOrders，图11-8显示了执行结果。

11

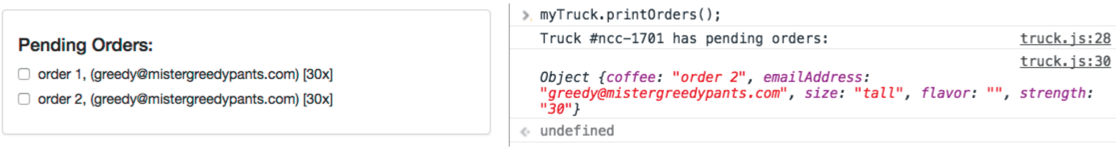


图11-8　UI中拥有相同邮箱地址的两个订单

之前说过：每个客户只能拥有一个公开订单。因为我们使用简单的键值对存储数据，所以同一邮箱地址对应订单中的后者会覆盖前者。正如控制台所显示的，“订单2”是唯一的待处理订单，“订单1”已经被覆盖了。

但是清单并没有反映出这一点，它依然显示“订单1”和“订单2”。在为订单添加一个清单行时，应该先确保之前所有与该邮箱地址相同的清单行都被删除了。

现在可以轻而易举地根据邮箱地址删除行。在checklist.js中更新addRow原型方法，使它要做的第一件事就是使用传入的邮箱地址作为参数来调用removeRow。

```
...
  CheckList.prototype.addRow = function (coffeeOrder) {
    //移除匹配相应邮箱地址的已有行
    this.removeRow(coffeeOrder.emailAddress);

    //使用咖啡订单信息创建一个新的Row实例
    var rowElement = new Row(coffeeOrder);

    //把新实例的$element属性添加到清单
    this.$element.append(rowElement.$element);
  };
...
```

保存checklist.js，并在浏览器中验证是否当具有相同邮箱地址的第二个订单被提交时，第一个订单的清单行已经被删除了。

既然已经可以从UI中删除待处理清单行了，那么就让我们专注于清单单击事件吧。

11.5.3　编写addClickHandler方法

我们将采用与FormHandler相同的注册事件处理程序来处理对清单的点击。

FormHandler.prototype.addSubmitHandler接受一个函数参数fn，然后我们注册一个匿名函数来处理this.$formElement的submit事件。在匿名函数的内部调用fn。下面是可以参考的方法定义：

```
FormHandler.prototype.addSubmitHandler = function (fn) {

  console.log('Setting submit handler for form');
  this.$formElement.on('submit', function (event) {
    event.preventDefault();

    var data = {};
    $(this).serializeArray().forEach(function (item) {
```

```
      data[item.name] = item.value;
      console.log(item.name + ' is ' + item.value);
    });
    console.log(data);

    fn(data);
    this.reset();
    this.elements[0].focus();
  });
};
```

这会使FormHandler.prototype.addSubmitHandler变得十分灵活，因为我们可以给它传递任何需要在表单提交之后被运行的函数。这样，FormHandler.prototype.addSubmitHandler既不需要知道该函数的详细信息，也不需要知道它所执行的步骤。

向CheckList添加一个名为addClickHandler的原型方法，该方法的工作方式与FormHandler的addSubmitHandler相同。换言之，它将进行以下操作。

(1) 接受一个函数参数。

(2) 注册事件处理程序回调。

(3) 在事件处理程序回调中调用第一步中的函数参数。

CheckList.prototype.addClickHandler 与 FormHandler.prototype.addSubmitHandler 不同的地方是，它将监听一个点击事件并将回调绑定到CheckList实例上。

在checklist.js中添加addClickHandler方法并指定一个名为fn的参数。采用jQuery的on方法监听点击事件。

在事件处理函数中声明一个名为email的本地变量，并将代表客户邮箱地址的event.target.value赋值给它。然后调用removeRow(email)，再调用fn(email)。记住，要使用bind(this)来设置事件处理程序函数的上下文对象。

```
...
  function CheckList(selector) {
    ...
  }

  CheckList.prototype.addClickHandler = function (fn) {
    this.$element.on('click', 'input', function (event) {
      var email = event.target.value;
      this.removeRow(email);
      fn(email);
    }.bind(this));
  };

  CheckList.prototype.addRow = function (coffeeOrder) {
    ...
```

使用this.$element.on注册回调事件处理程序时，要把click作为事件名称，但同时也要传入一个**过滤选择器**作为第二个参数。过滤选择器告知事件处理程序**当且仅当**事件是由<input>元素触发时才执行回调函数。

这种模式被称为**事件委托模式**，它的工作原理是因为click和keypress等事件都会通过DOM**传播**，这就意味着它们的祖先元素也会接收到事件。

每当要给待处理清单这类动态添加或者删除的元素绑定事件时，都应该使用事件委托的方式。为动态添加元素的容器添加单一事件监听程序，然后根据实际触发事件的元素执行相应的处理程序是更为容易、更加高效的做法。

请注意，**不要**在事件处理程序中调用event.preventDefault。为什么呢？因为如果调用event.preventDefault，该复选框将不会在被选中时改变其显示状态。

另外要注意，我们把事件处理程序回调绑定到了this，而它引用的是CheckList实例。

11.5.4　调用addClickHandler

addClickHandler需要关联到deliverOrder。回到main.js去进行关联——向checkList.addClickHandler传递一个绑定版的deliverOrder。

```
...
  var myTruck = new Truck('ncc-1701', new DataStore());
  window.myTruck = myTruck;
  var checkList = new CheckList(CHECKLIST_SELECTOR);
  checkList.addClickHandler(myTruck.deliverOrder.bind(myTruck));
  var formHandler = new FormHandler(FORM_SELECTOR);
...
```

保存修改，并在表单中添加一些咖啡订单。单击其中任何一个清单项的复选框或者文本，它将被删除（如图11-9所示）!

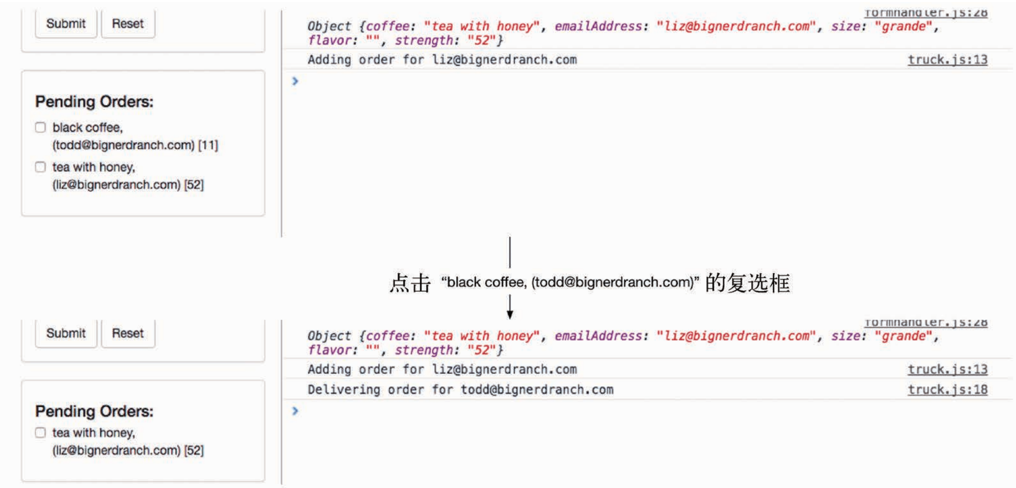


图11-9　单击一个清单项来删除它

我们现在已经学会了如何动态创建表单元素并使用它们的事件处理程序。可以将邮箱地址作为标识，将每个人与特定的咖啡订单相关联。

通过这些技术，我们完成了管理UI的模块，把一个只能在控制台中运行的程序转化为在现实中也可用的应用程序。

现在已经完成了CoffeeRun三个主要部分中的两个。除了用于管理应用程序数据的内部逻辑外，我们还添加了表单元素，使用FormHandler和CheckList来提供交互式的UI。接下来的几章将探讨如何与远端服务器交换数据。

11.6　初级挑战：在描述中加入浓度信息

假定咖啡的浓度是最为重要的信息，它应该位于描述的开始部分。

尝试更改订单描述的书写方式，让咖啡浓度位于描述文本的开头。

11.7　中级挑战：不同口味，不同颜色

尝试根据不同的口味显示不同颜色的订单。根据每种咖啡的选择情况，让清单中的不同行以不同的背景颜色显示。

记住要确保文本与背景颜色有足够强的对比度。

11.8　高级挑战：允许编辑订单

允许用户编辑现有的订单。你需要修改清单的工作方式。

如果用户双击某个订单，请将该订单重新载入表单进行编辑。如果用户只点击一次，则置灰该行，并在几秒钟之后视该订单项为已完成，并将其从清单和应用程序的数据中删除。

额外挑战：请确保用户在完成编辑之后，编辑后的行应该在原位置更新，而不是先删除再替换为新行。

第 12 章 表单校验

CoffeeRun已经可以运行了！用户可以在表单中输入咖啡订单的信息，这些信息之后会被处理并存储。但是思考一下，如果有人在表单中填写了错误的或者不可用的信息，那对应用和美食车会产生怎样的影响呢?

无须担心！通过代码可以很轻松地处理这种问题，确保应用所需数据的合法性。事实上，这是将数据发送给服务器前必须要做的事情。几乎每个现代浏览器都会在订单被提交前对其进行校验。我们所需要做的就是设置校验规则。

在本章中，我们将学习两种表单校验方式。第一种是在HTML上添加校验属性，使用浏览器内置的校验机制进行校验。第二种是使用约束校验API（Constraint Validation API），通过 JavaScript来编写我们自己的校验代码。

12.1 required 属性

最简单的表单校验是检测字段是否为空，这种校验对于带有默认值的字段不会生效，如杯型、口味、咖啡浓度。但订单和邮箱地址字段是必填的——你肯定不希望收到这些字段为空的订单。

在index.html中为订单和邮箱字段添加`required`属性，它是一个布尔属性。

```
...
          <div class="form-group">
            <label for="coffeeOrder">Order</label>
            <input class="form-control" name="coffee" id="coffeeOrder"
              autofocus required />
          </div>

          <div class="form-group">
            <label for="emailInput">Email</label>
            <input class="form-control" type="email" name="emailAddress"
            id="emailInput" value="" placeholder="dr@who.com"
              required />
          </div>
...
```

记住，无须为布尔属性赋值。如果误写为`required="false"`，事实上它的值仍然为`true`，这个字段仍是必填项！浏览器只关心这个属性是否**存在**，而不会关注它的值。

有必要再强调一遍：如果元素的某个属性是布尔属性，那么无论为它赋予什么值，浏览器均会认为该值为`true`。

保存index.html，确保browser-sync正在运行，在浏览器中打开CoffeeRun。尝试提交订单或者邮箱字段为空的表单，会看到一个警告，如图12-1所示。

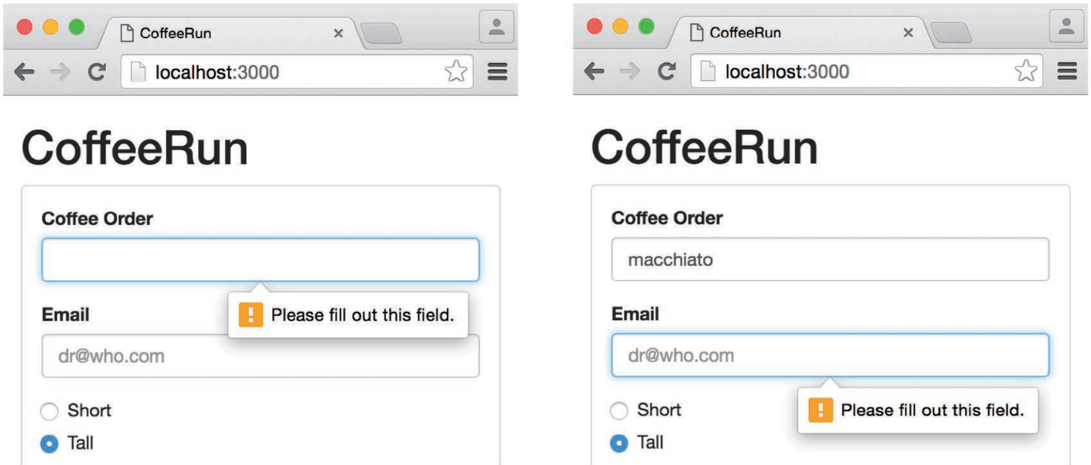


图12-1　当字段为空时抛出的错误

注意，提交处理程序没有在控制台中输出任何消息，因为`submit`事件只在浏览器校验表单通过后才会触发（如图12-2所示）。

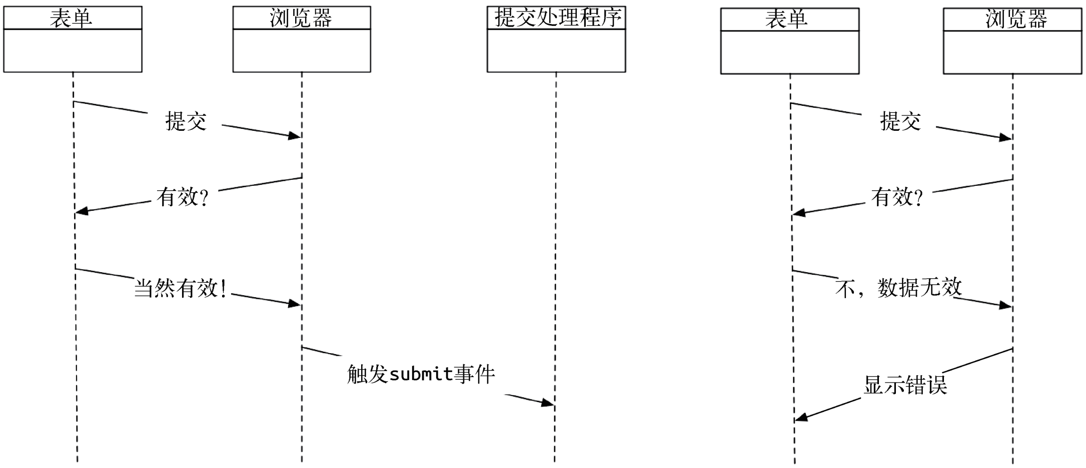


图12-2　表单验证时两个可能的事件序列

12.2　使用正则表达式校验表单

使用required属性是确保字段不为空的一种简单方式，但是如果我们希望限制表单字段的数据格式呢？这就需要使用pattern属性了。

在订单<input>标签中的required属性后添加pattern属性，并为其分配一个特殊格式的字符串。这种字符串被称为**正则表达式**，稍后会对它进行解释。

```
...
          <div class="form-group">
            <label for="coffeeOrder">Order</label>
            <input class="form-control" name="coffee" id="coffeeOrder"
              autofocus required pattern="[a-zA-Z\s]+" />
          </div>
...
```

正则表达式是一串用于模式匹配的字符。[a-zA-Z\s]+这句正则表达式的含义是：在由小写字母（a-z）、大写字母（A-Z）或者空白字符（\s）组成的集合中任选一个重复一次或多次（+）。

简单来说，当你提交表单时，这个字段只接受包含字母或空格的值。

保存并刷新页面，看看如果在订单字段输入符号或者数字并提交后会发生什么？

12.3　约束校验 API

在校验表单字段时，最健壮的做法是编写校验函数。你可以使用约束校验API来触发内置的校验行为。

但是这么做也要付出不小的代价——苹果的Safari浏览器对约束校验API的支持度非常低。

虽然有这些弊端，但对我们来说最好的做法是先编写符合标准的代码，然后通过JavaScript库去弥补浏览器的兼容性问题。（你可以从12.6节了解更多信息。）

假设美食车是面向我们公司的雇员的，因此我们需要保证客户都是我们的雇员。一个简单的办法是确保提交的邮箱地址都在我们公司的域名下。

可以对emailAddress字段使用pattern属性，但这是一个学习约束校验API的好机会。（其实还因为下一章会对校验函数进行扩展——不只是简单地校验邮箱域名，还需要发送到服务器进行查重校验。）

创建一个scripts/validation.js文件用于放置校验函数。在index.html上为新模块添加<script>标签。

```
...
      </div>
    </section>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.4/jquery.js"
      charset="utf-8"></script>
     <script src="scripts/validation.js" charset="utf-8"></script>
     <script src="scripts/checklist.js" charset="utf-8"></script>
     <script src="scripts/formhandler.js" charset="utf-8"></script>
     <script src="scripts/datastore.js" charset="utf-8"></script>
```

```
    <script src="scripts/truck.js" charset="utf-8"></script>
    <script src="scripts/main.js" charset="utf-8"></script>
  </body>
</html>
```

保存index.html。在validation.js中添加一个IIFE模块，创建一个空对象字面值，分配至变量`Validation`，并将该变量暴露到App命名空间。

```
(function (window) {
  'use strict';
  var App = window.App || {};

  var Validation = {

  };

  App.Validation = Validation;
  window.App = App;
})(window);
```

`Validation`模块只用于组织函数，所以它不需要写为构造函数。

在其中添加一个`isCompanyEmail`方法，该方法使用正则表达式校验邮箱地址并返回布尔值。（你可以随便修改邮箱域名。）

```
(function () {
  'use strict';
  var App = window.App || {};

  var Validation = {
    isCompanyEmail: function (email) {
      return /.+@bignerdranch\.com$/.test(email);
    }
  };

  App.Validation = Validation;
  window.App = App;
})(window);
```

将字符串放置在正斜杠（`//`）之间可以构成正则表达式。在斜杠之间，我们指定了一个字符串必须由一个或者多个字符（`.+`）和@bignerdranch.com组成。同时，还使用了反斜杠表示bignerdranch.com中的句点应被视为字面值。（一般来说，正则表达式中的句点可以用来匹配任意字符。）结尾的$表示字符串必须以@bignerdranch.com结尾；换句话说，在这之后不能有任何字符。

正则表达式的对象拥有一个`test`方法。把字符串传给`test`方法，它会返回一个布尔值——`true`表示字符串符合正则表达式的校验规则，而`false`表示不符合。（要了解更多关于正则表达式的知识，可以查看developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp。）

在控制台中测试一下`App.Validation.isCompanyEmail`函数（如图12-3所示）。

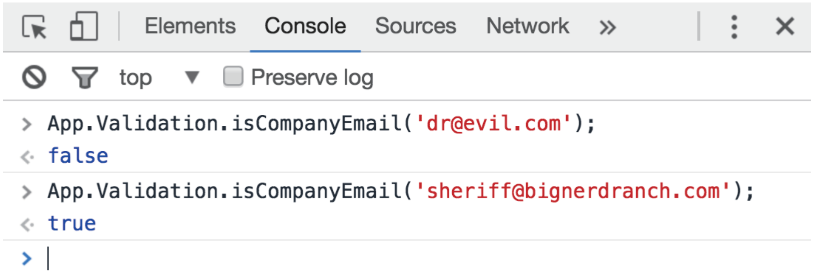


图12-3　在控制台测试App.Validation.isCompanyEmail

现在已经有了用于校验邮箱地址的的函数，接下来就可以将其和表单结合起来了。

12.3.1　监听input事件

当用户填写表单时，表单元素会触发各种事件。那什么时候使用这个校验函数呢？是在用户敲击字符时，或者在表单字段失去焦点时，又或者在提交表单时？

约束校验API需要在提交表单前标记其无效字段。只要有无效字段，浏览器就不会触发submit事件，所以在提交的时候才进行校验就太迟了。

在表单字段失去焦点时触发的事件被称作blur事件，这也不是一个校验的好时机。假设用户的光标在邮箱地址输入框中，即当前是这个字段获得焦点。如果用户输入完成后按下回车键，则会触发提交事件，而非失去焦点事件，所以相应的校验程序也不会执行。

因此只能在用户输入的时候执行校验程序。更新FormHandler.js中的FormHandler模块，在其中添加addInputHandler原型方法，它会在表单上绑定input事件监听器。和addSubmitHandler方法一样，它接受一个函数实参。

```
...
  FormHandler.prototype.addSubmitHandler = function (fn) {
    ...
  };

  FormHandler.prototype.addInputHandler = function (fn) {
    console.log('Setting input handler for form');
  };

  App.FormHandler = FormHandler;
  window.App = App;
...
```

使用jQuery的on方法绑定input事件的监听器，确保使用事件委托模式过滤掉了除[name="emailAddress"]字段外的其他字段触发的事件。

```
...
  FormHandler.prototype.addInputHandler = function (fn) {
    console.log('Setting input handler for form');
    this.$formElement.on('input', '[name="emailAddress"]', function (event) {
```

```
      //在此处插入事件处理程序
    });
  };

  App.FormHandler = FormHandler;
  window.App = App;
...
```

在事件处理程序中，从event.target对象中提取出邮箱地址字段的值。接着执行addInputHandler的函数参数fn，并且将邮箱地址作为参数传入，最后用console.log方法输出结果。

```
...
  FormHandler.prototype.addInputHandler = function (fn) {
    console.log('Setting input handler for form');
    this.$formElement.on('input', '[name="emailAddress"]', function (event) {
      // 在此处插入事件处理程序
      var emailAddress = event.target.value;
      console.log(fn(emailAddress));
    });
  };

  App.FormHandler = FormHandler;
  window.App = App;
...
```

保存formhandler.js。

12.3.2 将input事件和有效性校验绑定

在main.js中从App命名空间引入Validation，并将其保存到本地变量中。

```
...
  var Truck = App.Truck;
  var DataStore = App.DataStore;
  var FormHandler = App.FormHandler;
  var Validation = App.Validation;
  var CheckList = App.CheckList;
  var myTruck = new Truck('ncc-1701', new DataStore());
...
```

引入后，就可以将其和FormHandler的新方法addInputHandler结合使用。

在main.js的最后，将Validation.isCompanyEmail传入到formHandler实例的addInputHandler方法中。

```
...
  formHandler.addSubmitHandler(function (data) {
    myTruck.createOrder.call(myTruck, data);
    checkList.addRow.call(checkList, data);
  });

  formHandler.addInputHandler(Validation.isCompanyEmail);

})(window);
```

保存，然后刷新页面。填写邮箱地址，查看控制台的输出。在输入有效邮箱地址的过程中，控制台通过FormHandler.prototype.addInputHandler中的console.log(fn (email Address));输出一堆false。而当邮箱地址输入完成后，就会看到控制台输出true（如图12-4所示）。

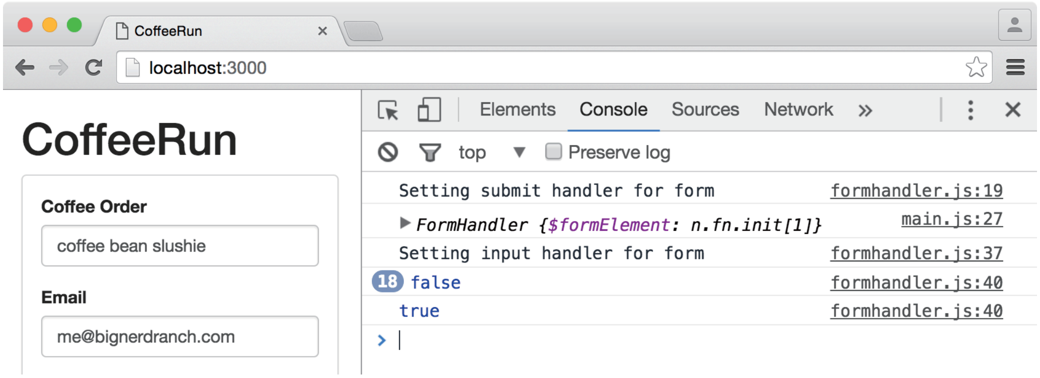


图12-4　输出邮箱校验结果

在敲击（或者删除）邮箱地址字段的每个字符的时候，校验程序都会执行。确认它能够正确校验输入后，就可以用它来展示错误信息了。

12.3.3　触发有效性检查

既然我们已经能验证邮箱地址是否属于我们公司域名下，就应该在检测失败时告知用户。使用event.target中的setCustomValidity方法将其标记为无效。

在formhandler.js中移除console.log语句。定义一个警告信息的变量，并添加if/else判断语句。如果fn(emailAddress)返回true，清除该字段的有效性提示，否则将警告信息分配给message变量，并将message设为有效性提示。

```
...
  FormHandler.prototype.addInputHandler = function (fn) {
    console.log('Setting input handler for form');
    this.$formElement.on('input', '[name="emailAddress"]', function (event) {
      var emailAddress = event.target.value;
      console.log(fn(emailAddress));
      var message = '';
      if (fn(emailAddress)) {
        event.target.setCustomValidity('');
      } else {
        message = emailAddress + ' is not an authorized email address!'
        event.target.setCustomValidity(message);
      }
    });
  };
...
```

被传入的的错误信息会展示给用户。所以即便没有错误，我们也要执行`setCustomValidity`，但是只需传入空字符串即可，这样可以将字段设置为有效。

在你输入时，有效性检测仅仅标记字段是否有效，而不展示错误信息。当你点击**提交**按钮时，浏览器会检测是否存在无效字段，并展示错误信息。

我们可以通过提交不合法的邮箱地址来进行测试——在点击**提交**按钮后，就会看到字段旁边作为警告出现的自定义提示消息（如图12-5所示）。

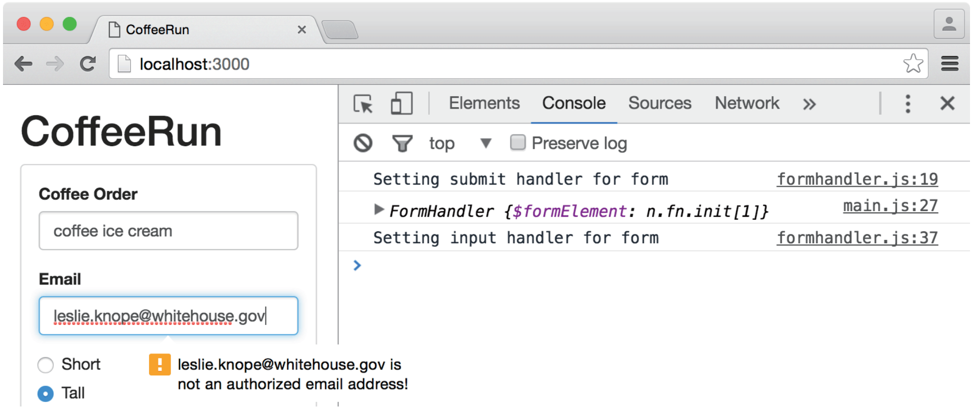


图12-5 只允许有效的邮箱地址

12.4 美化有效元素和无效元素

现在的CoffeeRun会检测订单字段和邮箱地址字段了。为了增强交互体验，来为无效字段加入视觉特效——只需要添加一小段CSS代码即可。在index.html的`<head>`标签中添加`<style>`标签。

```
...
  <head>
    <meta charset="utf-8">
    <title>coffeerun</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootst
rap/3.3.6/css/bootstrap.min.css">
    <style>
      form :invalid {
        border-color: #a94442;
      }
    </style>
  </head>
...
```

这会修改表单中拥有伪类`:invalid`的字段的边框颜色。这个伪类会在表单进行有效性检测时，由浏览器自动添加。

12

保存，然后回到浏览器。按几下Tab键（或者单击文本输入字段外的其他部分），聚焦在订单或者邮箱地址外的其他字段上。这时，这两个必填字段的边框会变成浅红色（如图12-6所示）。

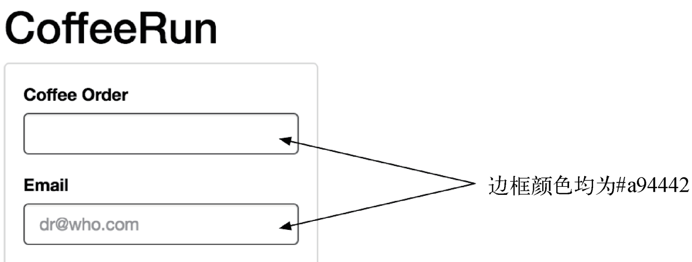


图12-6　相信我，这些边框是红色的

不过更好的做法是当必填字段获得焦点时才修改边框颜色，所以需要在index.html的选择器上再添加两个伪类。

```
  <head>
    <meta charset="utf-8">
    <title>coffeerun</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootst
rap/3.3.6/css/bootstrap.min.css">
    <style>
      form :focus:required:invalid {
        border-color: #a94442;
      }
    </style>
  </head>
...
```

这样，当且仅当元素拥有`:focus`、`:required`和`:invaild`三个伪类的时候边框才会变色（如图12-7所示）。

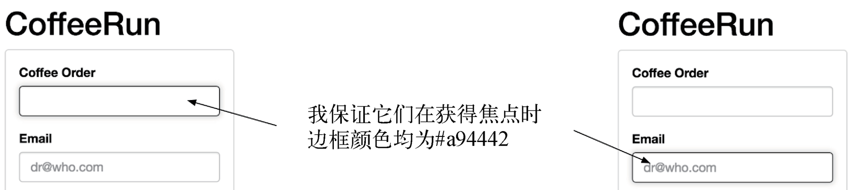


图12-7　只有在元素获得焦点时才展示无效的边框

CoffeeRun正逐渐发展为一个功能齐备的Web应用。在接下来的两章中，我们会通过Ajax与远端服务器同步数据。

12.5　中级挑战：为脱咖啡因咖啡进行自定义校验

在Validation模块中添加新函数。它会接受两个参数：一个字符串和一个整数。如果字符串包含decaf且数字大于20，则返回false。

为咖啡订单的文本字段和咖啡因浓度滑块添加事件监听器，并尝试在编辑字段时触发自定义校验，从而验证失败展示。

12.6　延展阅读：Webshim 库

我们之前提过，苹果的Safari浏览器并不支持约束校验API。如果我们需要使应用支持Safari，则需要使用一个库或者使用polyfill来模拟浏览器没有实现的API。

Webshim是一个能在Safari中提供自定义约束功能的库，可以从github.com/aFarkas/webshim下载它。

实际上，Webshim库可以polyfill许多功能。配置并使用它吧！（不过因为它做了很多事情，所以文档比较晦涩难懂。）

下面将学习如何使用它来帮助我们在Safari上使用Validation模块。首先，从项目github.com/aFarkas/webshim/releases/latest下载最新的压缩文件。解压文件，并将其放置到coffeerun目录下的 js-webshim/webshim文件夹中（就在index.html和scripts文件夹旁）。

在index.html上添加<script>引入webshim/polyfiller.js文件。

```
...
          </div>
        </section>
      <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.4/jquery.js"
        charset="utf-8"></script>
      <script src="webshim/polyfiller.js" charset="utf-8"></script>
      <script src="scripts/validation.js" charset="utf-8"></script>
      <script src="scripts/checklist.js" charset="utf-8"></script>
      <script src="scripts/formhandler.js" charset="utf-8"></script>
      <script src="scripts/datastore.js" charset="utf-8"></script>
      <script src="scripts/truck.js" charset="utf-8"></script>
      <script src="scripts/main.js" charset="utf-8"></script>
    </body>
</html>
```

然后在main.js中添加以下代码：

```
...
  var Validation = App.Validation;
  var CheckList = App.CheckList;
  var webshim = window.webshim;
  var myTruck = new Truck('ncc-1701', new DataStore());

  ...

  formHandler.addInputHandler(Validation.isCompanyEmail);
```

```
    webshim.polyfill('forms forms-ext');
    webshim.setOptions('forms', { addValidators: true, lazyCustomMessages: true });

}(window));
```

这样会引入webshim库并且在表单上使用它。

但是有一个比较奇怪的地方需要留意——在使用setCustomValidity的时候，必须使用jQuery包裹对象。在CoffeeRun中，则需要将formhandler.js中的addInputHandler方法里的event.target对象包裹起来。

```
...
  FormHandler.prototype.addInputHandler = function (fn) {
    console.log('Setting input handler for form');
    this.$formElement.on('input', '[name="emailAddress"]', function (event) {
      var emailAddress = event.target.value;
      var message = '';
      if (fn(emailAddress)) {
        $(event.target).setCustomValidity('');
      } else {
        message = emailAddress + ' is not an authorized email address!'
        $(event.target).setCustomValidity(message);
      }
    });
  };
...
```

Webshim的作者选择将polyfill功能完全实现为jQuery的扩展。除了这个包装，不需要对代码进行任何修改。

保存后，可以在Safari上进行测试。你会发现，在没有输入咖啡订单或者输入错误邮箱地址的情况下，也会有相应的提示出现。

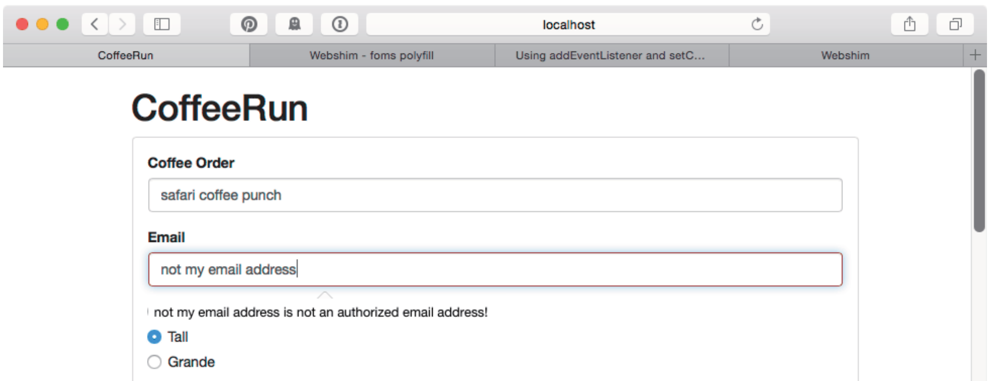


图12-8　使用Webshim作为Safari的polyfill

Webshim除了表单验证外还提供了很多功能，浏览文档看看你还可以做些什么。

第 13 章 Ajax

在上一章中，我们使用了浏览器内置的校验工具确保用户输入的数据符合CoffeeRun的要求。通过这个校验，我们大可放心地将数据发送到服务器。

此时，`FormHandler.prototype.addSubmitHandler`方法调用了事件对象的`preventDefault`方法来阻止浏览器向服务器发送请求。一般来说，服务器会返回一个促使页面重新加载的响应。然而，我们现在希望从表格中提取用户输入的数据，然后使用JavaScript更新表格和清单。

本章将创建`RemoteDataStrore`模块将请求发送到服务器并处理响应（如图13-1所示）。不过这项任务会在后台使用Ajax完成，这样的话就不需要重新加载页面了。

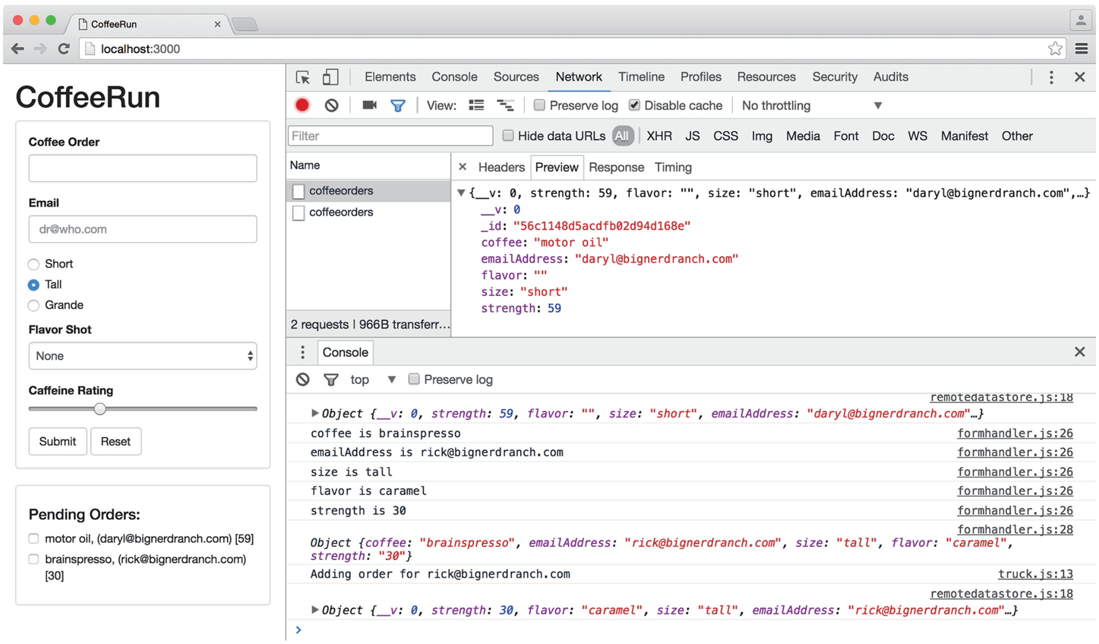


图13-1　本章结束后的CoffeeRun

Ajax是通过JavaScript与远端服务器通信的技术。JavaScript可以在无须刷新页面的情况下，利用服务器返回的数据更新页面，这大大改善了Web应用的体验。

最初，术语Ajax是asynchronous JavaScript and XML（异步JavaScript和XML）的缩写。但是现在无论是使用什么技术实现的，只要是符合这种风格的异步数据通信方式，都统称为Ajax。（异步通信意味着应用在发送请求后就可以执行其他任务，无需等待服务器响应。）如今，Ajax是在后台发送和接受数据的标准机制。

13.1　XMLHttpRequest 对象

Ajax的核心是XMLHttpRequest API。在现代浏览器中，我们可以实例化一个新的XMLHttpRequest对象。通过它能够在不刷新页面的情况下向服务器发送请求。这一切都在后台进行。

通过XMLHttpRequest对象，可以监听请求、响应周期中的任何阶段，这和在DOM对象上监听事件大同小异。我们还可以检查XMLHttpRequest的属性来获得现在的请求和响应周期的状态。其中response和status是两个十分有用的属性，它们会在改变发生的时候即时更新。response属性包括了服务器返回的数据（如HTML、XML、JSON等），status是用于展示HTTP响应是否成功的数字代码，它们也被称作**HTTP状态码**。

状态码按照范围分组，每个范围都有自己的基本含义。例如，200~299的状态码表示成功，而500~599的状态码表示服务器错误。这些范围通常会被称作"2*xx*"或者"3*xx*"状态。

表13-1展示了一些常用的状态码。

表13-1　常用HTTP状态码

状态码	状态文本	描　　述
200	OK	请求成功
400	错误请求	服务器不能理解该请求
404	找不到	找不到对应资源，通常是因为文件或路径名不正确
500	服务器内部错误	服务器遇到了一个错误，例如在服务器的代码中有一个未处理的异常
503	服务器不可用	服务器不能处理该请求，通常是因为服务器过载或者宕机

jQuery拥有许多创建和管理XMLHTTpRequest对象的方法，而且它还提供了一个简洁的、向后兼容的、跨浏览器的API。它不是唯一一个可以管理Ajax请求的库，但是许多库都遵循了它的设计。我们将会使用jQuery的get和post方法来进行Ajax的GET和POST请求，还会使用jQuery的ajax方法进行DELETE请求。

13.2　RESTful Web 服务

为了优化CoffeeRun，我们将使用远端Web服务器来存储应用数据。需要使用的服务器已经准备好了。

CoffeeRun服务器提供了RESTful Web服务。"REST"表示"具象状态传输"（representational state transfer），这是一种基于HTTP操作（GET、POST、PUT、DELETE）和用于标识服务器上资源的URL的Web服务风格。

通常，URL路径（服务器名称后的部分）会指代事物的集合（如/coffeeorders）或者通过ID标识个别事物（如/coffeeorders/[customer email]）。

URL的不同会影响HTTP操作的结果。例如，使用集合类的URL时，GET请求会检索集合中所有物品的列表；如果是使用个别事物的URL，GET请求会检索该事物的所有细节信息。

表13-2展示了URL和HTTP操作对应的结果。

表13-2 在RESTful规范下，不同的URL和HTTP操作得到的结果

URL 路径	GET	POST	PUT	DELETE
/coffeeorders	列举所有记录	创建一个记录	—	删除所有记录
/coffeeorders/a@b.com	得到该记录	—	更新记录	删除该记录

13.3 RemoteDataStore 模块

接下来会创建一个RemoteDataStore模块，它的任务是代表应用和服务器交流。RemoteDataStore模块将拥有和DataStore一样的方法——add、get、getAll和remove。我们将使用这些方法来和服务器进行交流（如图13-2所示）。

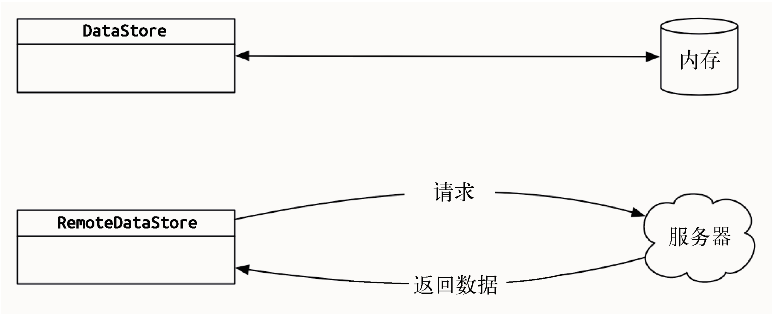


图13-2 DataStore VS RemoteDataStore

使用RemoteDataStore替代DataStore，这样就不需要更改Truck、FormHandler或者CheckList模块了。（你并不需要删除DataStore模块，以后会根据应用的在线情况在这两种存储方式间切换。）

RemoteDataStore的方法会在后台通过网络请求和服务器进行异步通信。当浏览器端接受服务器的返回后，便会执行回调函数。

每一个RemoteDataStore方法都接受一个函数实参，它会在接受服务器响应后执行。

创建一个scripts/remotedatastore.js文件，并且在index.html上为其添加一个<script>标签。

```
...
    <script src="scripts/validation.js" charset="utf-8"></script>
    <script src="scripts/checklist.js" charset="utf-8"></script>
    <script src="scripts/formhandler.js" charset="utf-8"></script>
    <script src="scripts/remotedatastore.js" charset="utf-8"></script>
```

```
    <script src="scripts/datastore.js" charset="utf-8"></script>
    <script src="scripts/truck.js" charset="utf-8"></script>
    <script src="scripts/main.js" charset="utf-8"></script>
  </body>
</html>
```

保存index.html。在remotedatastore.js中引入App命名空间和jQuery，然后创建IIFE模块和名为RemoteDataStore的构造函数。构造函数接受一个参数作为远端服务器的URL。如果这个参数并没有传递进来，构造函数会抛出一个错误。在模块定义的最后，将RemoteDataStore暴露到App命名空间上。

```
(function (window) {
  'use strict';
  var App = window.App || {};
  var $ = window.jQuery;

  function RemoteDataStore(url) {
    if (!url) {
      throw new Error('No remote URL supplied.');
    }

    this.serverUrl = url;
  }

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;

})(window);
```

13.4　向服务器发送数据

先编写add方法来将客户订单数据存储到远端Web服务器上。

为RemoteDataStore添加一个原型方法。和DataStore的add方法一样，它有key和val两个形参。虽然并不一定要使用相同的形参名，但是让它们保持一致是个好做法。

```
...
  function RemoteDataStore(url) {
    ...
  }

  RemoteDataStore.prototype.add = function (key, val) {
    // 放置要运行的代码
  };

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;
...
```

13.4.1　使用jQuery的$.post方法

我们会在RemoteDataStore模块中使用jQuery的$.post方法，这个方法会在后台像XMLHTTpRequest对象一样给服务器发送POST请求。

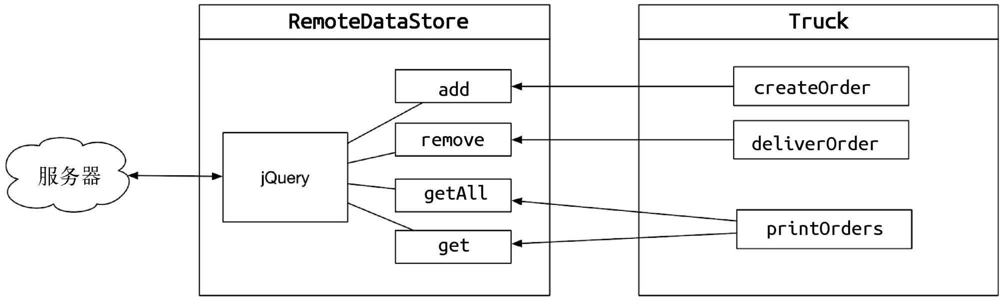


图13-3　RemoteDataStore使用jQuery进行Ajax请求

$.post方法只需要两个信息：服务器的URL和要带上的数据。

在remotedatastore.js中更新add方法，让其调用$.post方法，并传入this.serverUrl和val。

```
...
  RemoteDataStore.prototype.add = function (key, val) {
  // 放置要运行的代码
    $.post(this.serverUrl, val);
  };

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;
...
```

请注意，我们并没有使用key参数。添加它是为了保持add方法在RemoteDataStore和DataStore中的一致性——它们都将咖啡订单信息作为第二个参数。对RemoteDataStore来说，这是至关重要的一部分。

13.4.2　添加回调函数

和许多jQuery的方法一样，$.post接受一个可选的参数。传递一个回调函数作为第三个参数。当从服务器得到响应后，这个函数会被执行，并将数据传入其中。

这和我们编写的的事件处理代码十分相似——注册一个会在未来被调用的函数。处理事件时，这个时间点是鼠标点击或者提交表单的时候；处理远端数据时，这个时间点就是服务器响应返回的时候。

将一个匿名函数作为第三个参数传给$.post。这个匿名函数有一个形参，用server-Response来标记，并且使用console.log将其打印出来。

```
...
  RemoteDataStore.prototype.add = function (key, val) {
    $.post(this.serverUrl, val, function (serverResponse) {
      console.log(serverResponse);
    });
  };

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;
...
```

现在`$.post`知道了三件事情：和谁交流、交流什么和交流后要做什么。

保存修改内容，启动browser-sync并在浏览器中打开控制台。本书提供了测试服务器，可以利用它的URL实例化`RemoteDataStore`对象。（又一次，为了适应排版让代码折行了，请确保它们在同一行。）

```
var remoteDS = new App.RemoteDataStore
    ("http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders");
```

现在执行`add`方法，传入一些测试数据：

```
remoteDS.add('a@b.com', {emailAddress: 'a@b.com', coffee: 'espresso'});
```

在控制台中查看`console.log`语句输出的结果（如图13-4所示）。

```
> var remoteDS = new App.RemoteDataStore("http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders");
< undefined
> remoteDS.add('a@b.com', {emailAddress: 'a@b.com', coffee: 'espresso'});
< undefined
                                                                        remotedatastore.js:17
  Object {__v: 0, coffee: "espresso", emailAddress: "a@b.com", _id: "5712c496e36db403007d087c"}
>
```

图13-4　控制台展示`RemoteDataStore.add`的结果

控制台输出的对象里包含了服务器返回的信息：`coffee`和`emailAddress`信息，还有一些区分服务器的数据。

13.4.3　检查Ajax的请求和响应

点击开发者工具上方的NetWork（在Sources和Timeline之间），打开网络面板。这个面板展示了浏览器发出的请求，并且让我们查看它们的信息。

网络面板中可能会包括很多请求，点击左上角的⊘图标清除它们。然后激活下方的控制台抽屉，同时观看控制台和网络面板——只需按下键盘上的Esc键或者点击右上角的⋮图标，它会展示一个点击包含Show console选项的菜单，选中它后则会展示下方的控制台（如图13-6所示）。

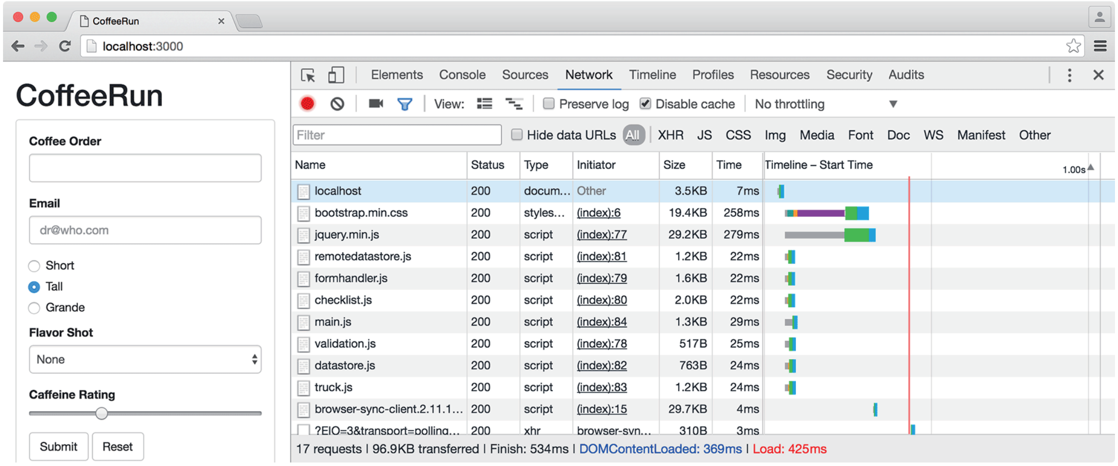


图13-5　在网络面板中查看Ajax请求

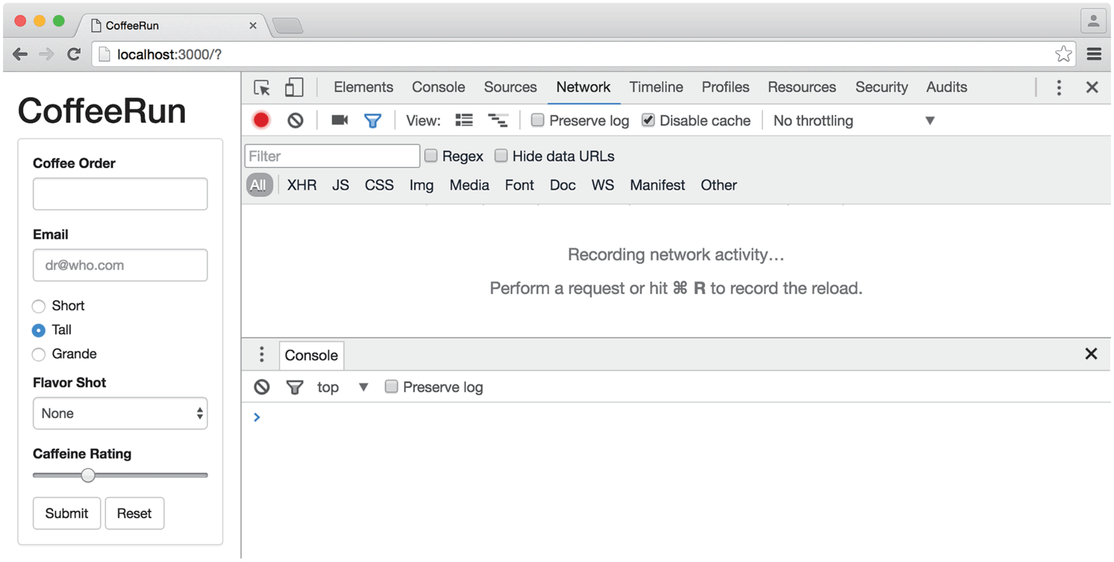


图13-6　在网络面板下方的控制台

在控制台中输入以下代码：

```
var remoteDS = new App.RemoteDataStore
    ("http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders");
remoteDS.add('a@b.com', {emailAddress: 'a@b.com', coffee: 'espresso'});
```

你会在网络面板上看到新的条目（如图13-7所示）。

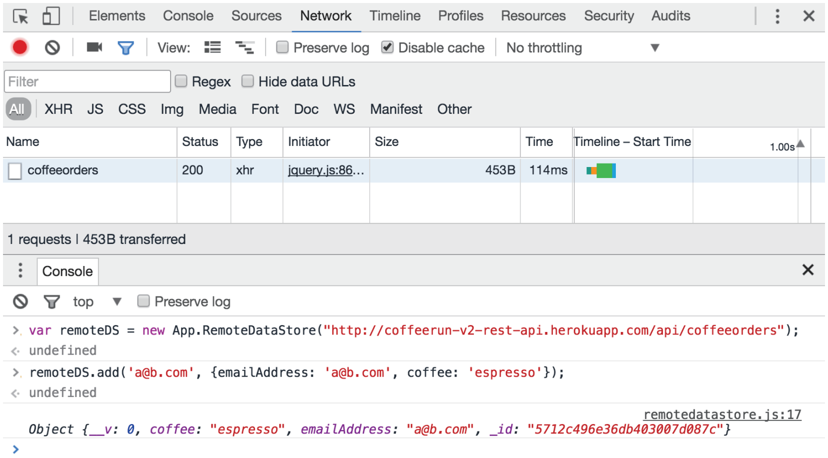


图13-7　网络面板中的Ajax请求

想了解与请求相关的更多信息，可以点击条目（如图13-8所示）。隐藏下方控制台，可以看到更多信息。

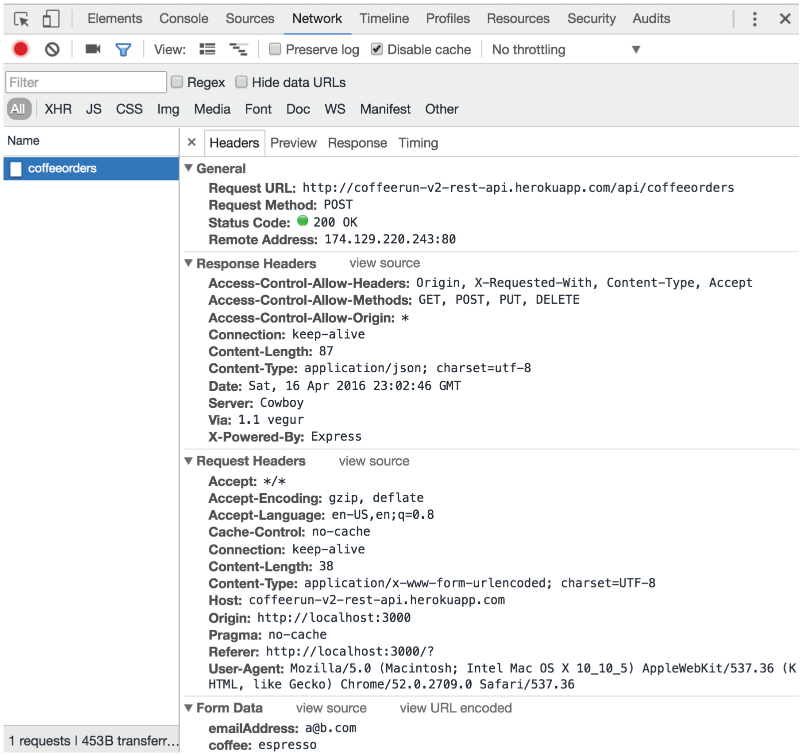


图13-8　请求的细节

详情的顶部是一些一般信息，底部是表格数据。中间则展示的是请求**首部**和响应**首部**，它们用于指定请求和响应的元数据和选项。

在这些信息中，状态码（在General窗格中）和表格数据窗格通常是在开发和调试Ajax请求时最有用的数据。

13.5 从服务器检索数据

现在的`RemoteDataStore`模块已经可以将独立的咖啡订单存储到服务器了，那么接下来就要添加`getAll`原型方法从服务器读取所有订单。开始修改remotedatastore.js吧！

```
...
  RemoteDataStore.prototype.add = function (key, val) {
    ...
  };

  RemoteDataStore.prototype.getAll = function () {
    // 放置要运行的代码
  };

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;
...
```

接下来要使用jQuery的`$.get`方法。和`$.post`一样，要对其传入服务器的URL。我们不需要传入数据，因为我们是要检索信息而非保存信息。除此之外，还要传入回调函数，这样就能在接收到服务端响应后进行调用。

在`RemoteDataStore.prototype.getAll`方法中调用`$.get`。

```
...
  RemoteDataStore.prototype.getAll = function () {
    // 放置要运行的代码
    $.get(this.serverUrl, function (serverResponse) {
      console.log(serverResponse);
    });
  };

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;
...
```

保存，切换到浏览器的开发者工具。

13.5.1 查看响应数据

和刚才一样，在控制台使用同样的URL实例化一个`RemoteDataStore`。（专业建议：为了不用每次都输入长长的URL，可以使用上下箭头切换刚刚输入在控制台中的语句。）然后调用`getAll`方法：

```
var remoteDS = new App.RemoteDataStore
    ("http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders");
remoteDS.getAll();
```

可以在开发者工具的网络面板上看到GET请求，它应该会被迅速响应，于是就能在控制台中看到一些咖啡订单信息了（之前输入到服务器的数据，如图13-9所示）。

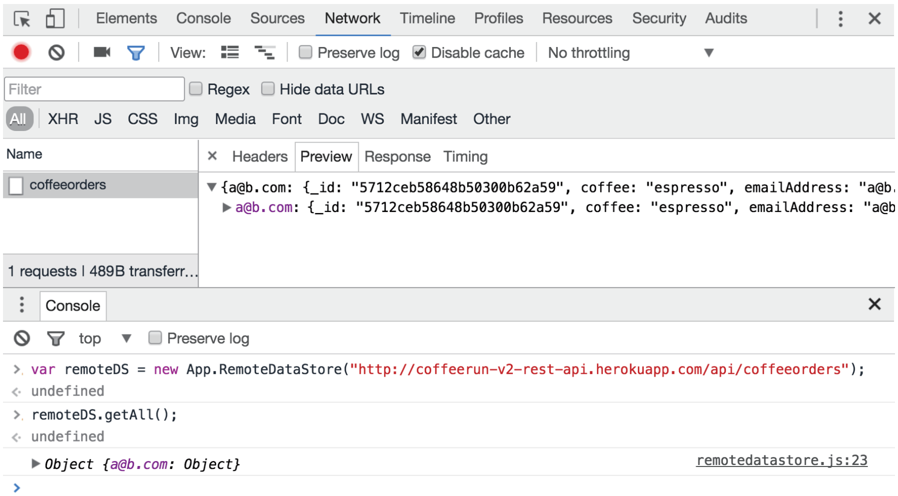


图13-9　查看getAll响应的数据

你看到的结果可能会有一些区别，这取决于你提交的信息。不过获得数据就意味着已经成功地从服务器检索到了数据。

13.5.2　添加回调函数

现在可以从服务器获取到数据，但是却不能从getAll返回相应的数据。这是因为getAll只是初始化了Ajax的请求，而没有处理响应。

不过我们给$.get传递了响应处理回调函数。这个回调函数和我们之前写过的事件处理回调函数一样——它们都期望接受一个参数，这意味响应数据只能在回调函数内部访问。那怎么在回调函数外部访问它呢？

如果给getAll传递一个函数实参，就可以在$.get的回调函数**内部**调用这个参数，也就能访问函数参数和服务器响应了。

添加或回调函数，并在remotedatastore.js中调用它。

```
...
  RemoteDataStore.prototype.getAll = function (cb) {
    $.get(this.serverUrl, function (serverResponse) {
      console.log(serverResponse);
      cb(serverResponse);
    });
```

```
  };

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;
...
```

getAll方法获得了远端服务器的所有订单数据，并且将数据传递给了它获得的回调函数cb。

我们还需要实现get方法，根据用户的邮箱地址获取单个咖啡订单。和getAll一样，它接受函数实参，同样用于传递获得的咖啡订单。

在remotedatastore.js中实现get：

```
...
  RemoteDataStore.prototype.getAll = function (cb) {
    ...
  };

  RemoteDataStore.prototype.get = function (key, cb) {
    $.get(this.serverUrl + '/' + key, function (serverResponse) {
      console.log(serverResponse);
      cb(serverResponse);
    });
  };

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;
...
```

保存remotedatastore.js。在控制台中输入以下代码，传入一个空的匿名函数给remoteDS.get。（它期望一个函数实参，但我们只想进行一次快速的测试。）

```
var remoteDS = new App.RemoteDataStore
    ("http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders");
remoteDS.get('a@b.com', function () {});
```

控制台应如图13-10所示。

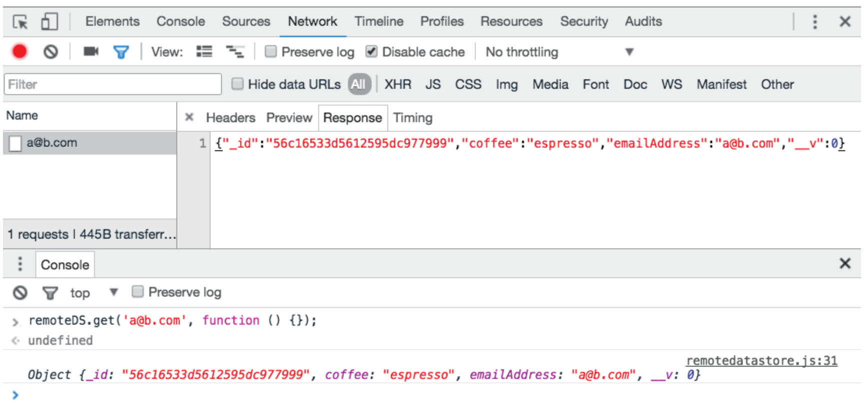


图13-10　测试RemoteDataStore.prototype.get

13.6　从服务器删除数据

通过Ajax，可以将订单保存到服务器上并查询订单，那么最后一件事就是要在订单完成后从服务器删除订单。

要完成这项任务，需要向单个订单的URL发送一个HTTP请求。就好像在RemoteDataStore.prototype.get中所做的一样，我们会使用服务器的URL，但还需要添加一个斜杠和用户邮箱地址。

向服务器发送一个DELETE请求。DELETE是一个HTTP操作，服务器在收到该请求后会知道你想删除与邮箱地址相关的数据。

使用jQuery的$.ajax方法

为了方便操作，jQuery提供了$.get和$.post这两个最常用的HTTP操作方法。当浏览器需要HTML、CSS、JavaScript或者图片文件时，就会使用GET请求；而提交表格的时候一般就会使用POST请求。

jQuery并没有提供一个通过Ajax发送DELETE请求的方便操作，因此我们使用$.ajax方法。（$.get和$.post本质上都是调用了$.ajax方法，并且注明了使用GET或者POST请求。）

在remotedatastore.js中添加remove的原型方法。在其中调用$.ajax方法，传入两个参数。第一个参数是一个咖啡订单的URL，由服务器URL、斜杠和键名（邮箱地址）组成；第二个参数是一个包括了Ajax请求的选项或设置的对象。你唯一需要为remove方法配置的就是将type设为DELETE。

```
...
  RemoteDataStore.prototype.get = function (key, cb) {
    ...
  };

  RemoteDataStore.prototype.remove = function (key) {
    $.ajax(this.serverUrl + '/' + key, {
      type: 'DELETE'
    });
  };

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;
...
```

（Ajax的请求有许多选项，你可以在api.jquery.com/jquery.ajax上了解更多。）

保存，然后回到控制台。实例化一个新的RemoteDataStore对象并执行remove方法，传入刚刚创建的测试订单的邮箱地址，最后调用getAll方法来检测订单是否被删除。

```
var remoteDS = new App.RemoteDataStore
    ("http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders");
remoteDS.remove('a@b.com');
remoteDS.getAll(function (data) { console.log(data); });
```

如果你查看服务器对DELETE请求的响应，会发现服务器返回了一段信息说明它干了什么（如图13-11所示）。（不过就像之前所说，不同的服务器可能会返回不同的信息。）

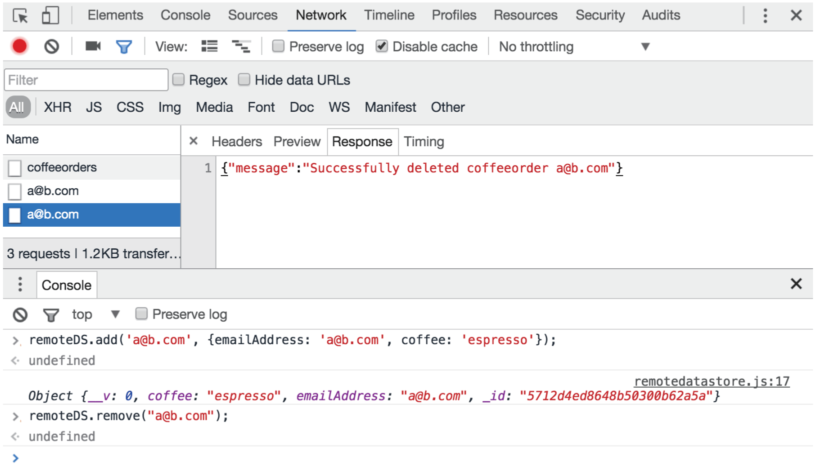


图13-11　检查DELETE请求的响应

13.7　用 RemoteDataStore 替换 DataStore

RemoteDataStore模块已经完成了，是时候用它替换掉DataStore了。

打开main.js。在App命名空间下引入RemoteDataStore。

```
(function (window) {
  'use strict';
  var FORM_SELECTOR = '[data-coffee-order="form"]';
  var CHECKLIST_SELECTOR = '[data-coffee-order="checklist"]';
  var App = window.App;
  var Truck = App.Truck;
  var DataStore = App.DataStore;
  var RemoteDataStore = App.RemoteDataStore;
  var FormHandler = App.FormHandler;
...
```

当然，还要添加一个名为SERVER_URL的新变量，并且将CoffeeRun测试服务器的URL分配给它。

```
(function (window) {
  'use strict';
  var FORM_SELECTOR = '[data-coffee-order="form"]';
  var CHECKLIST_SELECTOR = '[data-coffee-order="checklist"]';
  var SERVER_URL = 'http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders';
  var App = window.App;
  var Truck = App.Truck;
  var DataStore = App.DataStore;
  var RemoteDataStore = App.RemoteDataStore;
...
```

13

接下来，创建一个新的RemoteDataStore实例，并传入SERVER_URL。

```
...
  var RemoteDataStore = App.RemoteDataStore;
  var FormHandler = App.FormHandler;
  var Validation = App.Validation;
  var CheckList = App.CheckList;
  var remoteDS = new RemoteDataStore(SERVER_URL);
  var myTruck = new Truck('ncc-1701', new DataStore());
  window.myTruck = myTruck;
...
```

最后，将remoteDS，而不是DataStore实例，传入Truck构造函数。因为DataStore和RemoteDataStore拥有相同的方法，并且接受（几乎）一致的参数，这种变化将无缝工作。

```
...
  var RemoteDataStore = App.RemoteDataStore;
  var FormHandler = App.FormHandler;
  var Validation = App.Validation;
  var CheckList = App.CheckList;
  var remoteDS = new RemoteDataStore(SERVER_URL);
  var myTruck = new Truck('ncc-1701', new DataStore()); remoteDS);
  window.myTruck = myTruck;
...
```

保存修改，回到浏览器，输入一些咖啡订单信息并提交表单。这么做的时候记得要打开网络面板，那样就能看到在添加订单或者交付订单时网络请求的变化。

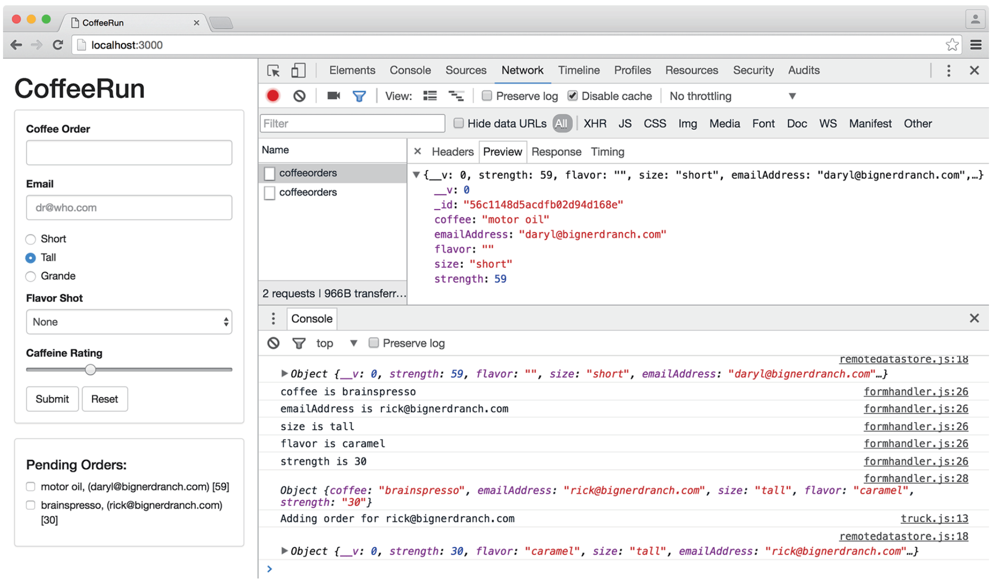


图13-12　将订单保存到远端服务器

恭喜！现在CoffeeRun已经能够正常运行并且和远端服务器进行交互了。

下一章是CoffeeRun的最后一章，虽然不会添加什么新的功能，但会更专注于重构代码，以学习异步代码的新模式。

13.8　中级挑战：校验远端服务器

我们的校验代码只是简单地做了域名检测。更新校验代码，让它可以检测这个邮箱地址是否被服务器中存储的订单使用过，以防提交了不合法订单。并且在收到不合法订单时，弹出一个校验警告。

你可能想打开两个浏览器窗口，然后输入不同的咖啡订单。

要注意校验时请求的发送频率。（可以在开发者工具的网络面板上查看该信息。）你有没有什么好方法来减少请求的数量呢？

13.9　延展阅读：Postman

Postman是用于发送测试请求的最佳工具之一，是一个免费的Chrome插件。它让我们在构建HTTP请求时，还能注明HTTP操作、表单数据、请求首部和用户凭据（如图13-13所示）。

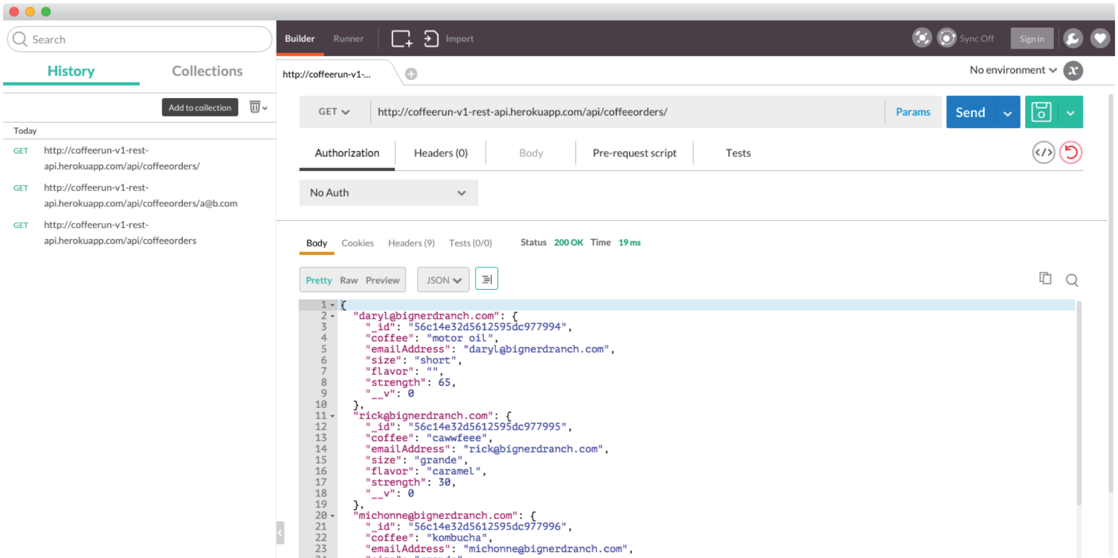


图13-13　Postman

Postman是在编写服务器通信代码之前探索API的不可或缺的工具。可以从Chrome的Web商店chrome.google.com/webstore中下载它（搜索“Postman”就可以找到它）。

第 14 章

Deferred和Promise

在CoffeeRun中模块化代码可以远离可怕的“面条式代码”（spaghetti code）——在将事件处理器（UI）代码和应用内部逻辑混合在一起使用的时候，特别容易出现这种代码。

我们的模块间通过函数进行交互，这种方式一般被称作回调。如果代码只依赖于一个单一的异步操作，回调是一种不错的解决方法。图14-1展示了一个CoffeeRun异步工作流的简化版。

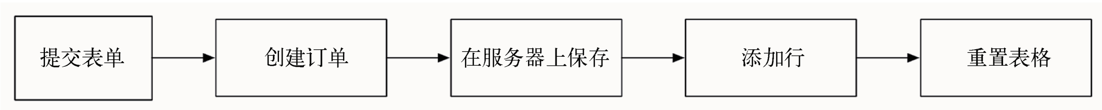


图14-1　在添加订单时的异步工作流

当有许多从属的异步操作时该怎么做呢？一种选择是嵌套回调函数，但很快我们就会发现这种方法既笨重又危险。如果想要编写一个简单但又必须带有错误处理的提交处理程序，最终结果很可能会是这样：

```
formHandler.addSubmitHandler(function (data) {
  try {
    myTruck.createOrder(function (error) {
      if (error) {
        throw new Exception(error)
    } else {
      try {
        saveOnServer(function (error) {
          if (error) {
            throw new Exception({message: 'server error'});
          } else {
            try {
              checkList.addRow();
            } catch (e2) {
              handleDomError(e2);
            }
          }
        })
      } catch (e) {
        handleServerError(e, function () {
          // 尝试再次添加行
          try {
```

```
                checkList.addRow();
              } catch (e3) {
                handleDomError(e3);
              }
            });
          }
        }
      });
    } catch (e) {
      alert('Something bad happened.');
    }
  });
```

在本章中，我们会学习更好的解决方式——Promise。相同的操作会通过Promise链进行表示，如下：

```
formHandler.addSubmitHandler()
  .then(myTruck.createOrder)
  .then(saveOnServer)
  .catch(handleServerError)
  .then(checkList.addRow)
  .catch(handleDomError);
```

Promise提供了一种管理复杂异步操作的方法。而在本章中，我们会学习如何使用它来优化CoffeeRun的架构。Promise是一个相对较新的功能，不过在最新的浏览器（包括Chrome）中都得到了很好的支持。

在CoffeeRun中，我们主要关注当前操作成功后，下一步操作是否执行。Promise简化了这一操作——不再依赖回调函数，而是通过返回Promise对象，一步步解耦模块。

14.1 Promise 和 Deferred

Promise对象有三种状态：pending、fulfilled和rejected（如图14-2所示）。

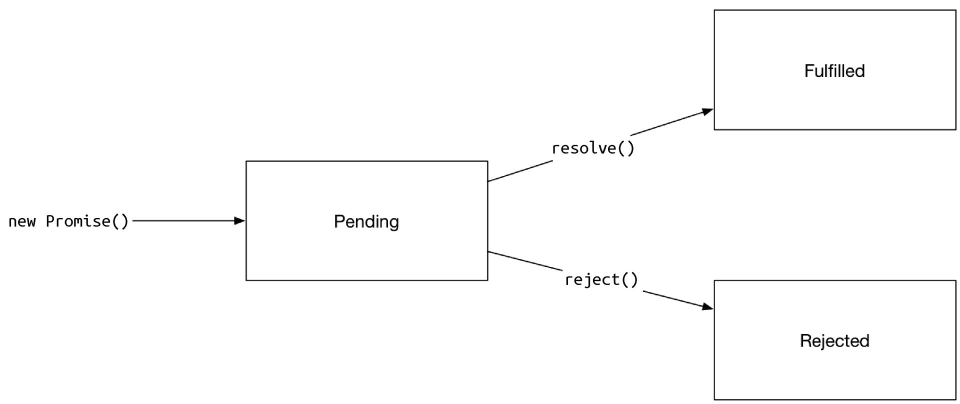


图14-2 Promise对象的三种状态

每个Promise对象都有一个then方法，它会在Promise状态变为fulfilled时被触发。我们可以调用then并且传入一个回调函数。当Promise的状态变为fulfilled时，回调函数就会被执行，同时会收到Promise异步操作后所取得的数据。

可以链式调用多个then函数。相比于编写函数，再让其接受数据并调用回调函数，返回一个Promise对象以利用then作为链式调用更为妥当。

先从jQuery的Deferred对象开始，它和Promise十分相似。

jQuery的$.ajax方法（包括$.post和$.get方法）会返回一个Deferred对象。Deferred对象会根据自身的两种状态——fulfilled和rejected——分别调用对应函数。从更新RemoteDataStore模块开始，这样子它就能返回由jQuery的Ajax方法生成的Deferred。接着，修改其他模块，让它们在Deferred上注册回调函数。

14.2　返回 Deferred

现在，开始使用jQuery的$.ajax方法返回的Deferred对象。在remotedatastore.js中更新原型方法，让它将$.get、$.post、$.ajax的执行结果返回如下：

```
...
  RemoteDataStore.prototype.add = function (key, val) {
    return $.post(this.serverUrl, val, function (serverResponse) {
      console.log(serverResponse);
    });
  };

  RemoteDataStore.prototype.getAll = function (cb) {
    return $.get(this.serverUrl, function (serverResponse) {
      console.log(serverResponse);
      cb(serverResponse);
    });
  };

  RemoteDataStore.prototype.get = function (key, cb) {
    return $.get(this.serverUrl + '/' + key, function (serverResponse) {
      console.log(serverResponse);
      cb(serverResponse);
    });
  };

  RemoteDataStore.prototype.remove = function (key) {
    return $.ajax(this.serverUrl + '/' + key, {
      type: 'DELETE'
    });
  };
...
```

因为它们现在返回由jQuery的Ajax方法生成的Deferred对象，所以get和getAll并不一定要接受回调函数。为了确认是否有回调函数，需要在执行cb前添加if语句进行判断。

```
...
  RemoteDataStore.prototype.getAll = function (cb) {
    return $.get(this.serverUrl, function (serverResponse) {
      if (cb) {
        console.log(serverResponse);
        cb(serverResponse);
      }
    });
  };

  RemoteDataStore.prototype.get = function (key, cb) {
    return $.get(this.serverUrl + '/' + key, function (serverResponse) {
      if (cb) {
        console.log(serverResponse);
        cb(serverResponse);
      }
    });
  };
...
```

保存remotedatastore.js。既然`RemoteDataStore`的方法会返回`Deferred`，我们就也需要更新`Truck`方法。首先，处理`createOrder`和`deliverOrder`。

打开truck.js，然后在这两个方法中调用`this.db`的语句前添加`return`。

```
...
  Truck.prototype.createOrder = function (order) {
    console.log('Adding order for ' + order.emailAddress);
    return this.db.add(order.emailAddress, order);
  };

  Truck.prototype.deliverOrder = function (customerId) {
    console.log('Delivering order for ' + customerId);
    return this.db.remove(customerId);
  };
...
```

保存truck.js。`Truck`现在会将`RemoteDataStore`生成的`Deferred`返回。在使用`Promise`和`Deferred`时，最好的做法就是将它们返回，这能让调用`createOrder`或者`deliverOrder`的对象所注册的回调函数在异步操作完成后被调用。

下一节就是这么做的。

14.3 通过 then 注册回调函数

`$.ajax`返回了一个`Deferred`，它拥有`then`方法。`then`方法用于注册一个当`Deferred`被解析（`resolved`）时调用的回调函数。当回调函数执行时，它会收到从浏览器返回的相应数据。

先从`then`开始。在main.js中，提交处理程序调用了`createOrder`和`addRow`方法。改变它们，以便让`addRow`成为`createOrder`的一个注册回调的方法。

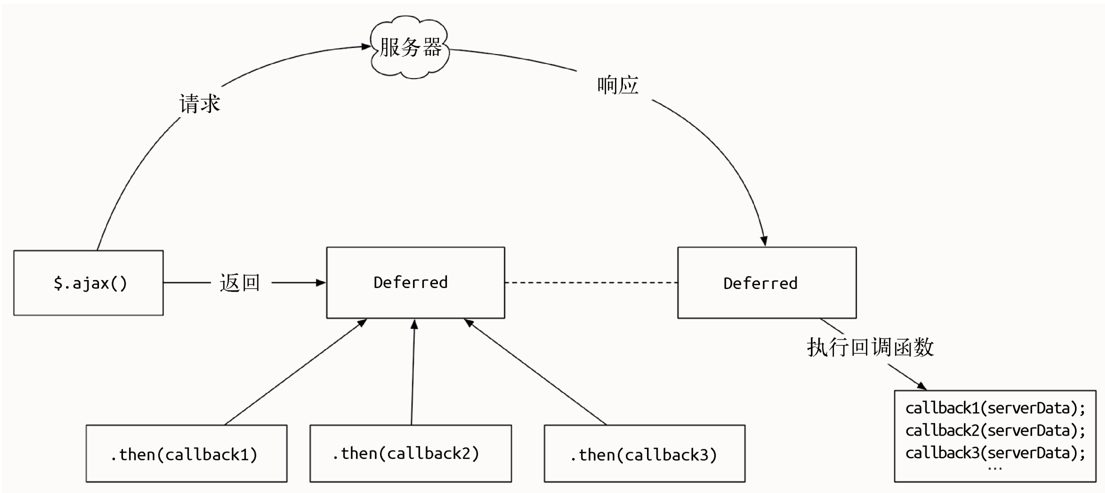


图14-3　Deferred对象执行使用then注册的回调函数

打开main.js，然后更新formHandler.addSubmitHandler的回调函数。在createOrder后添加.then。传入一个执行checkList.addRow的回调函数。

```
...
  formHandler.addSubmitHandler(function (data) {
    myTruck.createOrder.call(myTruck, data);
      .then(function () {
        checkList.addRow.call(checkList, data);
      });
  });
...
```

这让addRow在createOrder正常运行完毕后才被调用，而不是立即被调用。

14.4　使用 then 处理失败的情况

then接受第二个参数，它会在Deferred切换到rejected状态时执行。在main.js中的formHandler.addSubmitHandler上添加第二个函数实参（记得在两个参数间添加逗号）来查看效果。在函数内部会弹出带有错误信息的警报。

```
...
  formHandler.addSubmitHandler(function (data) {
    myTruck.createOrder.call(myTruck, data)
      .then(function () {
        checkList.addRow.call(checkList, data);
      },
      function () {
        alert('Server unreachable. Try again later.');
      }
      );
  });
...
```

在main.js的顶部故意拼错服务器的名称，这样子Ajax就会失败。（这个改变只是暂时的，所以只需要剪切URL的一部分，待会再粘贴回去就好。）

```
(function (window) {
  'use strict';
  var FORM_SELECTOR = '[data-coffee-order="form"]';
  var CHECKLIST_SELECTOR = '[data-coffee-order="checklist"]';
  var SERVER_URL = 'http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders/';
  var App = window.App;
...
```

保存更改，确保browser-sync正在运行。然后在浏览器中打开CoffeeRun，填写表单，在提交时会看到一个错误弹窗。

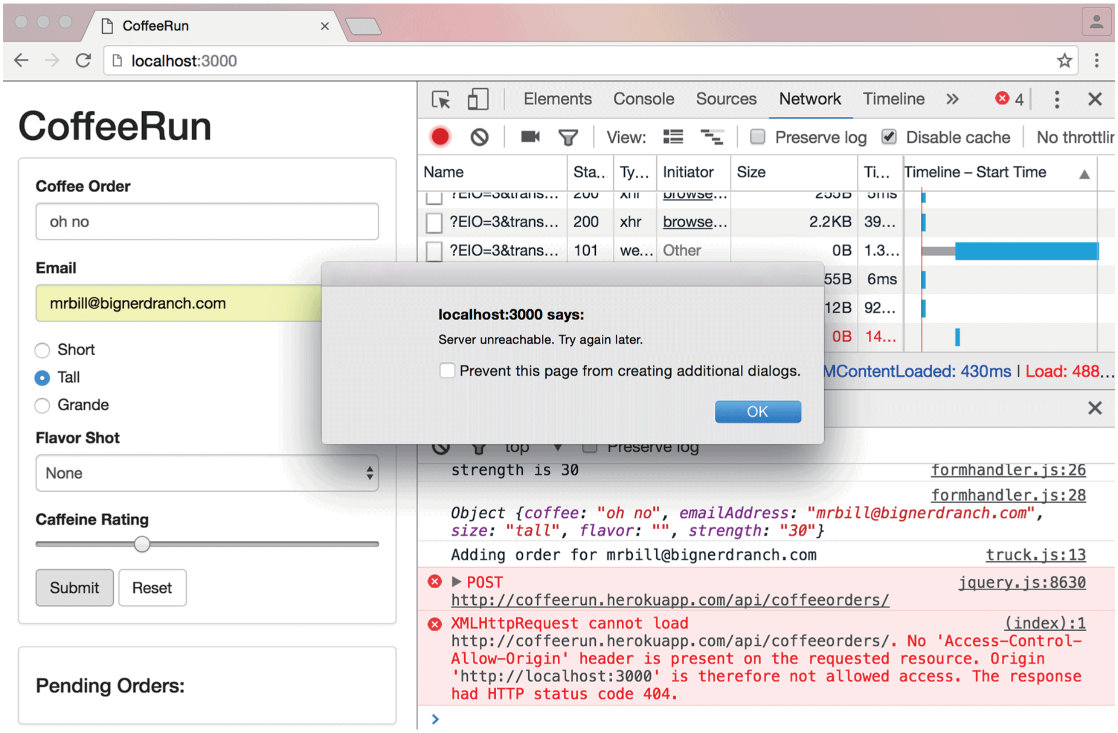


图14-4　当Ajax失败的时候弹出警告

将SERVER_URL恢复为http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders/。当然，也可以删除弹出警告的函数。

```
(function (window) {
  'use strict';
  var FORM_SELECTOR = '[data-coffee-order="form"]';
  var CHECKLIST_SELECTOR = '[data-coffee-order="checklist"]';
  var SERVER_URL = 'http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders/';
```

```
  ...

  formHandler.addSubmitHandler(function (data) {
    myTruck.createOrder.call(myTruck, data)
      .then(function () {
        checkList.addRow.call(checkList, data);
      },
      function () {
        alert('Server unreachable. Try again later.');
      }
      );
  });
...
```

使用then注册的回调函数与Promise的工作方式相对应。如果Promise的状态变为fulfilled，将会执行第一个回调；如果变为rejected，就会执行第二个回调。

14.5　在仅支持回调函数的 API 上使用 Deferred

有时候要让基于Deferred的代码和只支持回调函数的API接口协同合作——例如事件监听器。

目前，无论Ajax请求是否成功，formHandler.addSubmitHandler都会重置表单并且聚焦到第一个元素上。然而，我们只希望在Ajax请求成功的时候才这么做；换言之，我们只想在Deferred状态变为fulfilled的时候才如此执行。

那怎么知道Deferred何时会变为fulfilled呢？addSubmitHandler的函数实参会返回一个Deferred，你可以在addSubmitHandler内部的Deferred后添加一个.then。

在main.js中的回调函数中添加return关键字。

```
...
  formHandler.addSubmitHandler(function (data) {
    return myTruck.createOrder.call(myTruck, data)
      .then(function () {
        checkList.addRow.call(checkList, data);
      });
  });
...
```

保存main.js，然后打开formhandler.js，找到addSubmitHandler方法，定位到相应的匿名函数fn。因为现在匿名函数会返回一个Deferred，所以可以在其后添加一个.then。使用.then注册一个回调函数用于重置表单并聚焦到第一个元素。

```
...
  FormHandler.prototype.addSubmitHandler = function (fn) {
    console.log('Setting submit handler for form');
    this.$formElement.on('submit', function (event) {
      event.preventDefault();

      var data = {};
```

```
      $(this).serializeArray().forEach(function (item) {
        data[item.name] = item.value;
        console.log(item.name + ' is ' + item.value);
      });
      console.log(data);
      fn(data);
      .then(function () {
        this.reset();
        this.elements[0].focus();
      });
    });
  };
...
```

之前，我们有三个顺序语句：执行回调、重置表单，聚焦到第一个元素。现在，我们的语句会依赖于前一个语句的返回结果。我们执行了回调，则**当且仅当**执行正常且没有任何异常时，才会**继续**重置表单并且聚焦到第一个函数。

这时候还有一个小问题。当使用`.then`注册回调函数时，它会有一个新的作用域，因此需要在匿名函数上使用`.bind`为`FormHandler`实例设置它的`this`的值。

修改一下formhandler.js。

```
...
  FormHandler.prototype.addSubmitHandler = function (fn) {
    ...
      fn(data)
        .then(function () {
          this.reset();
          this.elements[0].focus();
        }.bind(this));
    });
  };
...
```

保存formhandler.js。

同样，我们也只希望在`Truck.prototype.deliverOrder`成功时才从清单中删除选项。

因此在checklist.js中的`addClickHandler`后添加`.then`。记住，要通过`.bind`为匿名函数绑定`this`的值。

```
...
  CheckList.prototype.addClickHandler = function (fn) {
    this.$element.on('click', 'input', function (event) {
      var email = event.target.value;
      this.removeRow(email);
      fn(email);
        .then(function () {
          this.removeRow(email);
        }.bind(this));
    }.bind(this));
  };
...
```

保存checklist.js。回想一下，我们是在main.js中调用`addClickHandler`的：

```
checkList.addClickHandler(myTruck.deliverOrder.bind(myTruck);
```

不需要对这个方法进行改动，因为`Truck.prototype.deliverOrder`会自动返回`Deferred`，`addClickHandler`会如它所写的那样工作。

因为数据都是远程的，所以需要加载数据并为每个咖啡订单绘制清单。可以使用`Truck.prototype.printOrders`和`CheckList.prototype.addRow`来达成这个目标。

需要修改`printOrders`的两个地方。首先，要将`printOrders`更新为可以使用`Deferred`；接着，为其添加一个函数实参，用于遍历需要打印的数据。

在truck.js中的`Truck.prototype.printOrders`目前看起来是这样的：

```
...
  Truck.prototype.printOrders = function () {
    var customerIdArray = Object.keys(this.db.getAll());

    console.log('Truck #' + this.truckId + ' has pending orders:');
    customerIdArray.forEach(function (id) {
      console.log(this.db.get(id));
    }.bind(this));
  };
...
```

将代码更新一下，返回`this.db.getAll`，然后在其后调用`.then`。在`.then`中传入一个匿名函数，并且用`.bind`进行`this`绑定。

```
...
  Truck.prototype.printOrders = function () {
    return this.db.getAll()
      .then(function (orders) {
        var customerIdArray = Object.keys(this.db.getAll());

        console.log('Truck #' + this.truckId + ' has pending orders:');
        customerIdArray.forEach(function (id) {
          console.log(this.db.get(id));
        }.bind(this));
      }.bind(this));
  };
...
```

我们的匿名函数期望接受包含服务器返回的所有咖啡订单信息的对象。抽取对象中的所有键名，并且将它们分配给变量`customerIdArray`。

```
...
  Truck.prototype.printOrders = function () {
    return this.db.getAll()
      .then(function (orders) {
        var customerIdArray = Object.keys(this.db.getAll()); orders);

        console.log('Truck #' + this.truckId + ' has pending orders:');
        customerIdArray.forEach(function (id) {
          console.log(this.db.get(id));
```

```
        }.bind(this));
      }.bind(this));
  };
...
```

同样，改变console.log语句，让它不再调用this.db.get(id)，而是使用包含所有咖啡订单信息的orders对象。我们不应该在打印每个订单的时候都进行Ajax请求。

```
...
  Truck.prototype.printOrders = function () {
    return this.db.getAll()
      .then(function (orders) {
        var customerIdArray = Object.keys(orders);

        console.log('Truck #' + this.truckId + ' has pending orders:');
        customerIdArray.forEach(function (id) {
          console.log(this.db.get(id)); orders[id]);
        }.bind(this));
      }.bind(this));
  };
...
```

printOrders应该接受一个可选的函数实参，所以我们需要判断到底是否有函数被传入。如果有，则执行。在执行的时候，传入目前的所有订单orders[id]。

```
...
  Truck.prototype.printOrders = function (printFn) {
    return this.db.getAll()
      .then(function (orders) {
        var customerIdArray = Object.keys(orders);

        console.log('Truck #' + this.truckId + ' has pending orders:');
        customerIdArray.forEach(function (id) {
          console.log(orders[id]);
          if (printFn) {
            printFn(orders[id]);
          }
        }.bind(this));
      }.bind(this));
  };
...
```

保存truck.js。在main.js中执行printOrders，然后传入checkList.addRow。记住，要确保addRow绑定在CheckList的实例上。

```
...
  formHandler.addInputHandler(Validation.isCompanyEmail);

  myTruck.printOrders(checkList.addRow.bind(checkList));
})(window);
```

保存，然后回到浏览器，CoffeeRun应该会展示出清单里存在的咖啡订单。手动刷新页面来确认清单是否每次都被重新填充，查看网络面板来确认Ajax请求是否发生（如图14-5所示）。

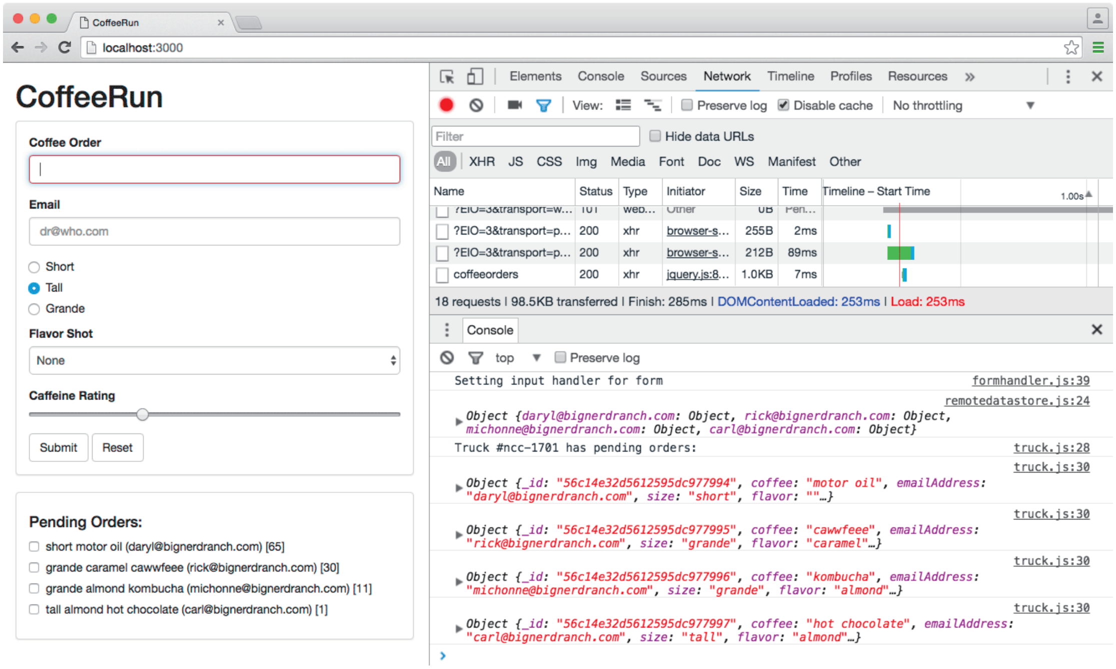


图14-5　在页面加载的时候绘制订单

14.6　为 DataStore 配置 Promise

通过用RemoteDataStore的方法返回Deferred，你可以更灵活地使用从服务器拉取的数据。

不过你会注意到，RemoteDataStore的方法和DataStore的方法间存在较大的差异。如果切回到DataStore，应用就不能正常工作了。

在图14-6中可以看到，使用常规DataStore实例化Truck会抛出错误，并且不能和UI正确交互。CoffeeRun期望一个基于Promise的DataStore来进行工作。

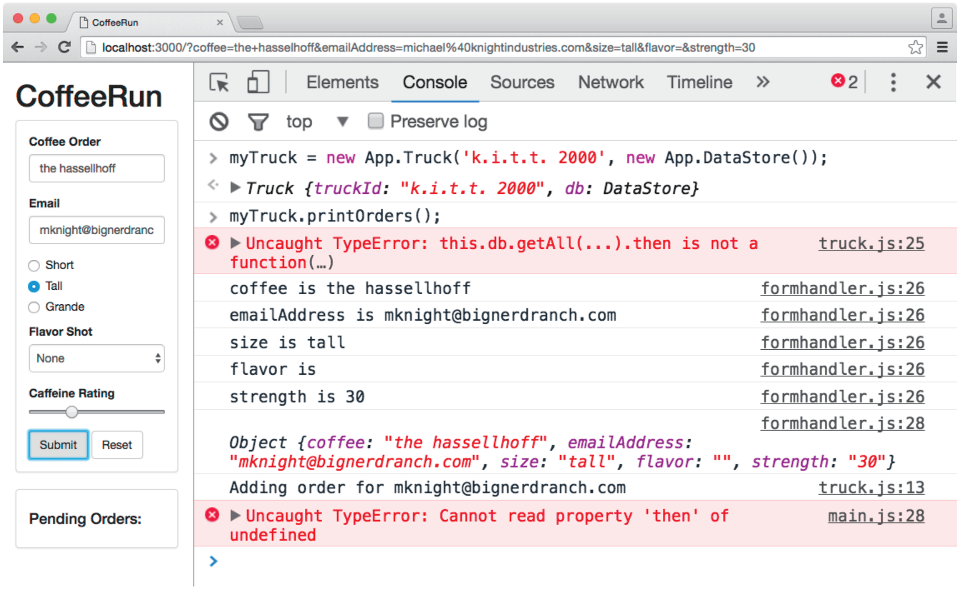


图14-6　DataStore不再兼容

要补救这种状况，需要修改DataStore的4个方法，让它们返回Promise。

我们已经使用过jQuery的Deffered对象，然而因为DataStore并没有使用jQuery的$.ajax方法，所以需要使用原生的Promise构造函数来创建并返回Promise。

14.6.1　创建并返回Promise

在datastore.js中先更新add方法。首先，创建一个Promise变量，然后赋予它window.Promise。尽管这不是必须的，但将全局空间里所需的部分引入到我们自己的模块中是一个挺好的做法。

在add方法中创建一个promise变量，并分配一个Promise实例。确保在最后返回这个实例。

```
(function (window) {
  'use strict';
  var App = window.App || {};
  var Promise = window.Promise;

  function DataStore() {
    this.data = {};
  }

  DataStore.prototype.add = function (key, val) {
    this.data[key] = val;
    var promise = new Promise();

    return promise;
  };
...
```

Promise的构造函数需要函数实参。传入一个带有resolve和reject两个形参的函数。

```
...
  DataStore.prototype.add = function (key, val) {
    this.data[key] = val;
    var promise = new Promise(function (resolve, reject) {
    });

    return promise;
  };
...
```

当Promise运行的时候，它会执行匿名函数并传入两个值：resolve和reject。执行 resolve函数会将Promise的状态变为fulfilled，而执行reject函数会将其状态变为rejected。

然后，将存储数据的代码（this.data[key] = val;）移动到匿名函数内。为了确保this.data正确指代DataStore的实例的data属性，在匿名函数后绑定this。

```
...
  DataStore.prototype.add = function (key, val) {
    this.data[key] = val;
    var promise = new Promise(function (resolve, reject) {
      this.data[key] = val;
    }.bind(this));

    return promise;
  };
...
```

14.6.2　resolve一个Promise

在匿名函数的最后执行resovle，不需要传入实参。

```
...
  DataStore.prototype.add = function (key, val) {
    var promise = new Promise(function (resolve, reject) {
      this.data[key] = val;
      resolve(null);
    }.bind(this));

    return promise;
  };
...
```

为什么使用null作为实参呢？往DataStore添加值并不会产生一个值，所以不需要返回数值。当不需要返回数值时，应该使用null明确表示。（也可以使用resolve(val)让接下来的函数可以获取最新存储的数值，但这对于CoffeeRun没有必要，因此在本例中就不引入了。）

14.6.3　将其他DataStore方法Promise化

可以手动更新其余3个方法为相同的模式。但是相比于重新编写这些代码，创建一个名为

promiseResolvedWith的辅助函数来生成一个Promise似乎更方便。可以使用这个辅助函数更新DataStore.prototype.add。

```
...
  function DataStore() {
    this.data = {};
  }

  function promiseResolvedWith(value) {
    var promise = new Promise(function (resolve, reject) {
      resolve(value);
    });
    return promise;
  }

  DataStore.prototype.add = function (key, val) {
    var promise = new Promise(function (resolve, reject) {
      this.data[key] = val;
      resolve(null);
    }.bind(this));

    return promise;
    return promiseResolvedWith(null);
  };
...
```

promiseResolvedWith是我们编写add方法时所使用的Promise代码的可复用结构。它接受一个名为value的参数，然后创建一个名为promise的变量，并且分配一个Promise实例。它将一个带有resolve和reject两个形参的匿名函数传入Promise的构造函数里。在匿名函数内，执行resolve并将value值传入。

不需要在promiseResolvedWith中将函数实参的this进行绑定，因为在其内部并没有引用到this。

将其余方法更新为使用promiseResolvedWith。在get和getAll中将先前版本的值传入，而在remove方法中传入null。

```
...
  DataStore.prototype.get = function (key) {
    return this.data[key];
    return promiseResolvedWith(this.data[key]);
  };

  DataStore.prototype.getAll = function () {
    return this.data;
    return promiseResolvedWith(this.data);
  };

  DataStore.prototype.remove = function (key) {
```

```
      delete this.data[key];
      return promiseResolvedWith(null);
    };
  ...
```

最后，更新main.js，用`DataStore`替代`RemoteDataStore`。

```
  ...
    var remoteDS = new RemoteDataStore(SERVER_URL);
    var myTruck = new Truck('ncc-1701', remoteDS); new DataStore());
    window.myTruck = myTruck;
  ...
```

完成上述更改后，保存代码，再次测试一下CoffeeRun。你会看到它使用`DataStore`正常运行，并且不会发出任何Ajax请求。（如图14-7所示）

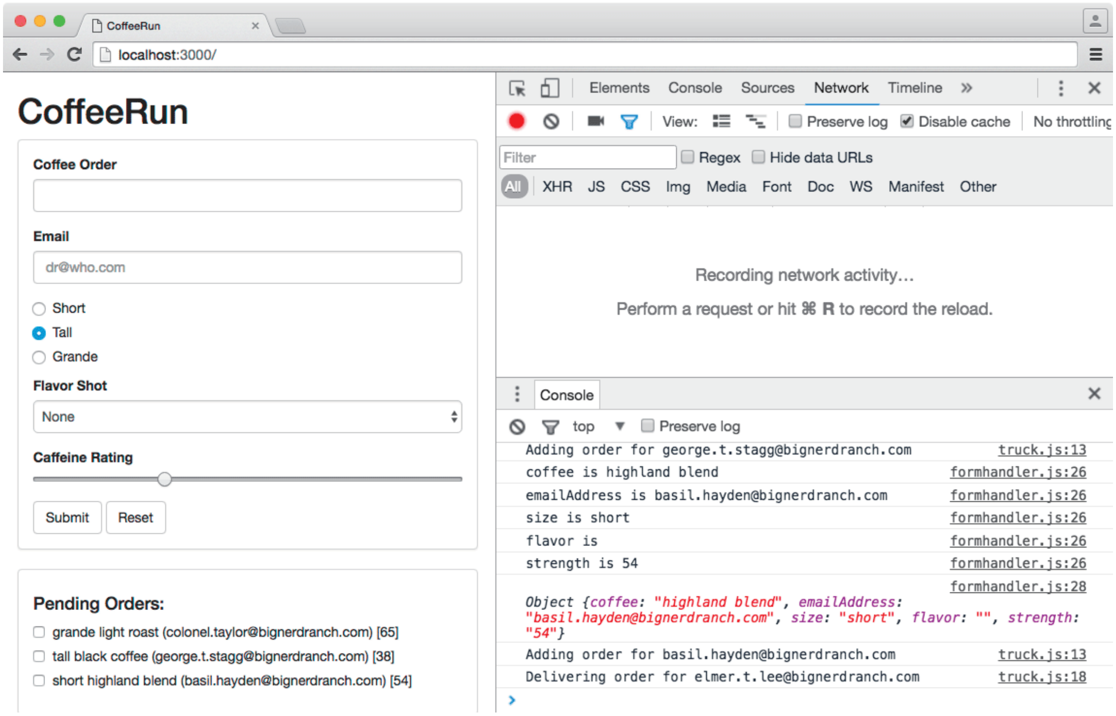


图14-7　完成CoffeeRun

CoffeeRun已经陪伴了你一段时光。在这段日子里，你使用IIFE、回调函数和`Promise`编写了一些严格的JavaScript代码，也使用jQuery操作了DOM元素，并与RESTful Web服务器进行了交互。

所以是时候放下CoffeeRun，踏出下一步了。下一个应用——Chattrbox——是一个**全栈**聊天应用。你在编写前端代码的同时还要编写服务器。不要因为这是你的第一个服务器应用而感到过分焦虑，你仍然会使用JavaScript，只是不在浏览器中而已。让我们准备好来使用Node.js吧。

14.7　中级挑战：回退到 Datastore

如果你足够幸运，你总是会有靠谱的网络。但是你要做好在使用CoffeeRun的时候网络中断的准备。

更新CoffeeRun，让它能在Ajax请求失败的时候使用`DataStore`。

要检测代码是否正常运行，在加载和保存咖啡订单的时候关闭电脑网络。

第三部分

实时数据传输

第 15 章

Node.js入门

15

Node.js是一个开源项目，能够让JavaScript代码在浏览器之外运行。

浏览器端的JavaScript代码可以访问`document`和`window`这样的全局对象以及其他API和库函数；Node环境下的JavaScript代码则能够访问硬盘驱动器、数据库和网络（如图15-1所示）。

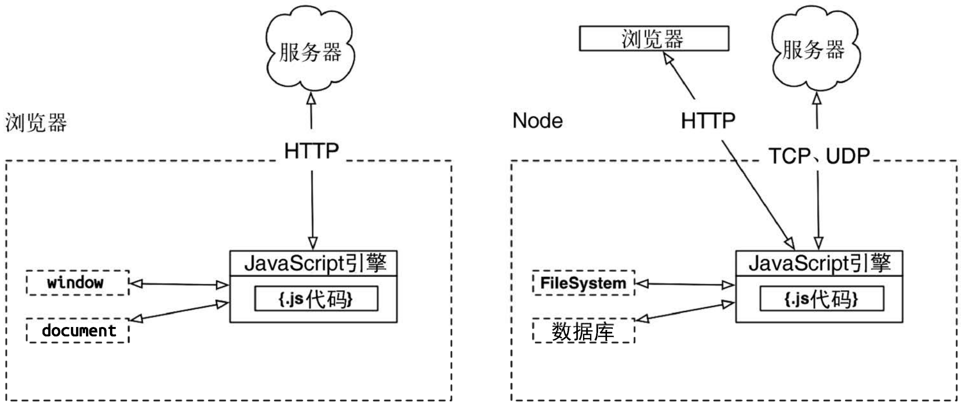


图15-1　在浏览器里运行的JavaScript与通过Node运行的JavaScript

使用Node能够创建各种应用，下至命令行工具，上至Web服务器。接下来的4章会用Node创建名为Chattrbox的实时聊天应用（如图15-2所示）。

Chattrbox由两部分组成：Node.js服务器和浏览器端的JavaScript应用。浏览器会连接到Node服务器，获取HTML、CSS和JavaScript文件。随后，JavaScript应用便会通过WebSocket处理实时通信。整个流程如图15-3所示。

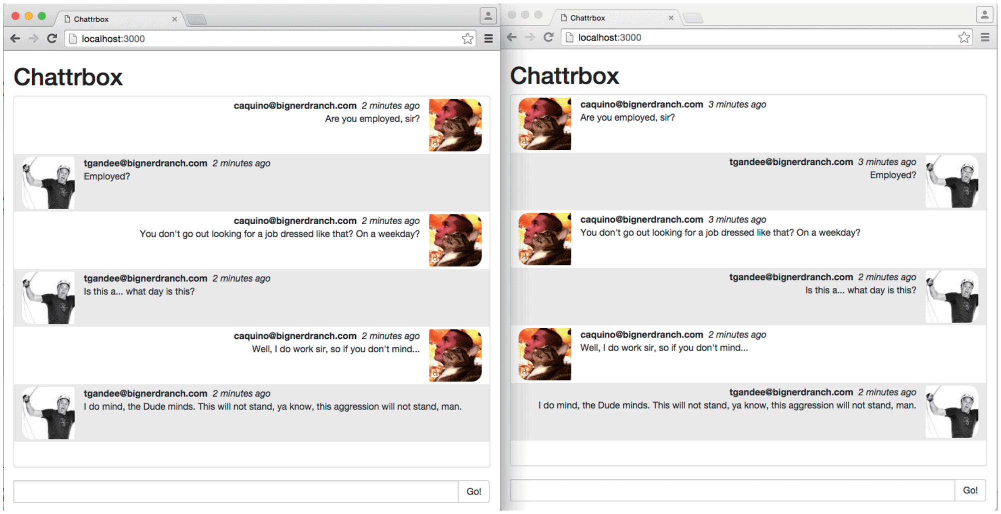


图15-2　Chattrbox：完全可用于重要会话的应用

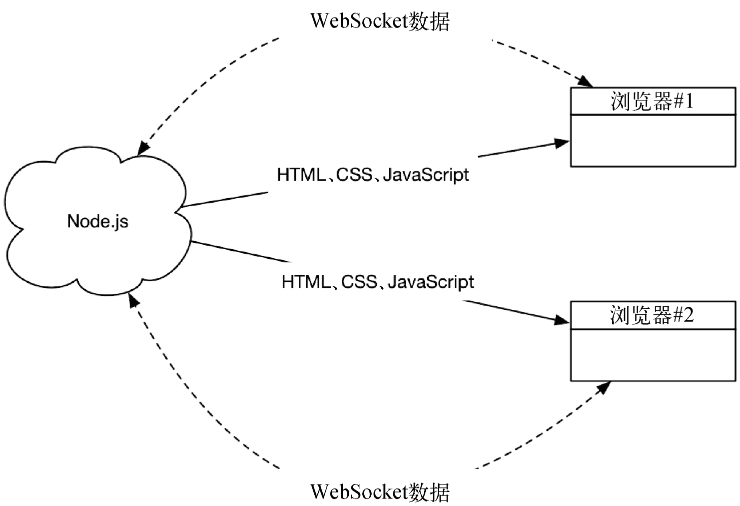


图15-3　Chattrbox应用的网络示意图

下一章会详细介绍WebSocket，本章的主要任务是熟悉Node。

15.1　Node 和 npm

因为第1章中已安装好Node.js，所以现在可以直接使用node和npm这两个命令行程序。npm可用于安装开源工具，如browser-sync；而node则负责运行JavaScript程序。

本章中的大部分工作都要用到npm。npm命令行工具能执行各种任务，如安装项目依赖、管理项目工作流和外部依赖。本章将会使用npm来完成如下任务。

- 使用`npm init`创建package.json文件。
- 使用`npm install --save`添加第三方模块。
- 运行保存在package.json的`scripts`中的常用命令。

除node和npm命令行外，Node还有很多好用的模块，这些模块提供的构造函数能够访问文件和文件夹、进行网络通信、处理事件等。此外，在Node中运行的JavaScript代码还能访问一些公用函数，这些函数增强了JavaScript与Node模块生态系统之间的交互。举个例子，Node提供了比CoffeeRun中的IIFE更简单的模块模式。

上面提到的package.json文件相当于Node项目的配置文件，它包含项目名称、版本号、描述等信息。更重要的是，它还能存储测试和应用构建时用到的配置和命令。

可以手动创建该文件，但用npm自动生成会更加方便。

15.1.1　`npm init`

在项目文件夹下创建chattrbox目录。打开终端进入该目录，运行`npm init`创建package.json。

npm会询问项目相关信息，同时也会提供默认值。目前使用默认值就够了，按回车键确认即可（如图15-4所示）。

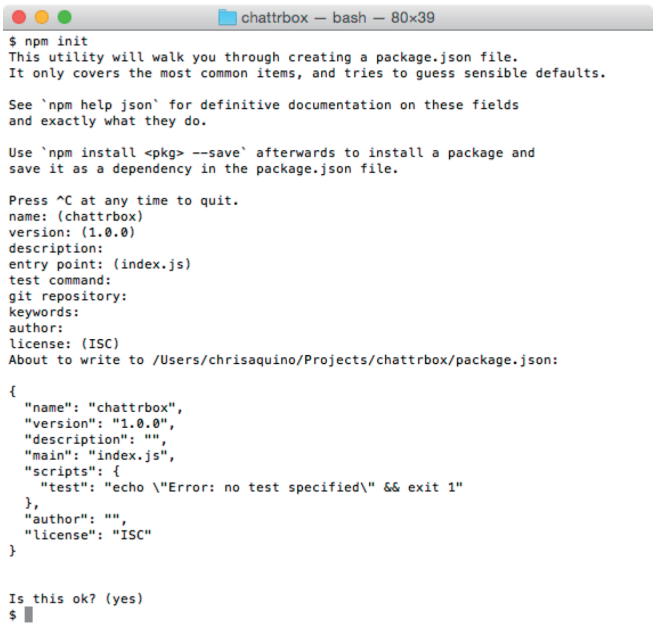


图15-4　运行`npm init`

用Atom打开项目文件夹，可以看到package.json文件已经创建好了。

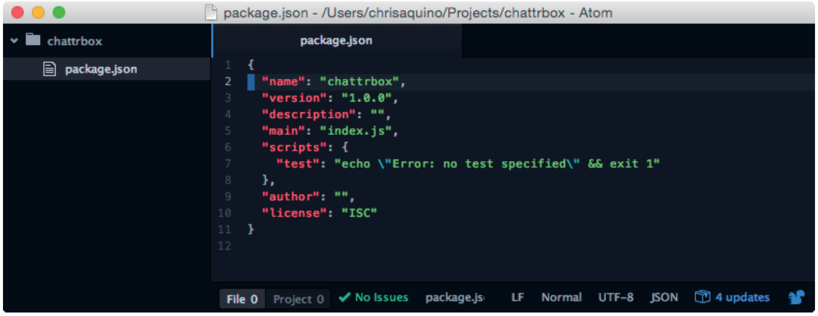


图15-5　执行`npm init`之后生成的package.json内容

15.1.2　npm脚本

在package.json中有一个`"scripts"`字段，该字段用于存储开发过程中可能会多次用到的命令。

构建Chattrbox时，可以在package.json的`"scripts"`字段添加命令，以提高开发效率。首先，添加一个`"start"`脚本，这是我们创建的第一个npm工作流脚本（别忘了在`"test"`一行的末尾添加一个逗号）：

```
...
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node index.js"
  },
...
```

添加好上面的脚本命令后，在命令行运行`npm start`即可启动Node服务。

15.2　Hello, World

为了解释运行于浏览器之外的JavaScript，首先来写一个经典的“Hello, World”程序。在chattrbox文件夹里新建index.js文件并输入以下代码，稍后会对代码进行解释。

```
var http = require('http');

var server = http.createServer(function (req, res) {
  console.log('Responding to a request.');
  res.end('<h1>Hello, World</h1>');
});
server.listen(3000);
```

第一行代码使用Node内置的`require`函数访问Node中的`http`模块，该模块提供了很多用来处理HTTP请求和响应的工具，如`http.createServer`函数。

`http.createServer`接受一个函数作为唯一的参数，每次HTTP请求时都会调用该函数（参数）。这种方式很像浏览器中的事件回调模式，只不过它是服务器端事件（接受一个HTTP请求）触发的回调。

回调函数在控制台打印了一条消息，并在响应中写入一些HTML文本。在Node中通常使用`req`和`res`作为HTTP请求和响应对象的变量名。

最后，使用`server.listen`让服务器监听3000端口，这一过程通常叫作“端口绑定”。

保存文件。运行命令`npm start`验证Node服务器的效果，终端结果如图15-6所示。

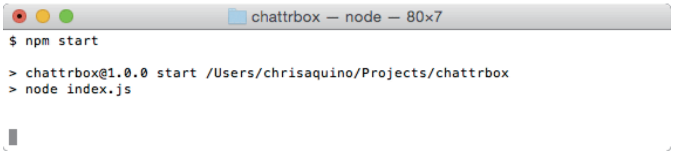


图15-6　通过`npm start`运行index.js

接着打开浏览器，输入http://localhost:3000，结果如图15-7所示。（注意，在Chrome之外的某些浏览器中可能会看到纯文本的HTML。这些浏览器可能需要一个`doctype`，或者需要从响应中读取额外的元数据，才能将响应当作HTML解析。你将要在本章末尾的挑战中解决这一问题。）

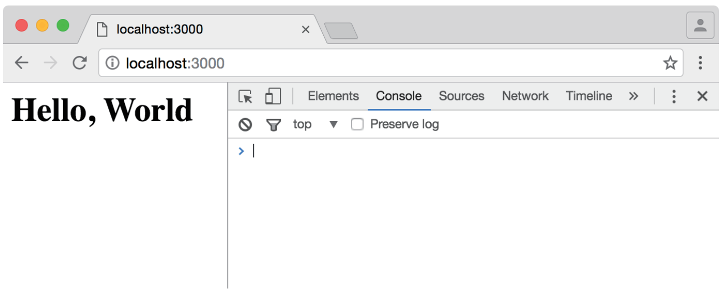


图15-7　在浏览器中访问Node服务器

和Ottergram和CoffeeRun不同的是，在浏览器端并不会看到JavaScript。看到这个页面时，服务端JavaScript代码已完成所有任务。

回到终端。可以看到，收到请求时，`console.log`函数打印出了`Responding to a request`（如图15-8所示）。

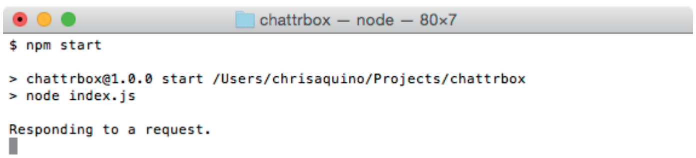


图15-8　请求到达时`console.log`

15.3　添加一个 npm 脚本

除编写命令行JavaScript程序之外，Node还提供了在开发过程中组织工作流的方法——一定要好好利用这一强大功能。接下来往项目里增加一些自动化的东西，看看它是如何工作的。

以运行服务器来举例。在每次修改代码时，你都要重复以下操作。

- 在编辑器里修改代码。
- 切换到终端。
- 按Control + C停止程序。
- 运行`npm start`再次启动程序。

可以通过编程的方式实现自动重启服务。幸运的是，已经有人替我们写好了程序，这是一个名为`nodemon`的模块。尽早在工作流里加入`nodemon`可以让编程体验更加流畅。

在终端停止程序并运行以下命令来安装`nodemon`模块：

```
npm install --save-dev nodemon
```

接着会出现如下几行提示，这是npm在提示package.json文件里有一些空字段。不必惊慌，明白npm对细节十分严格就好。

```
npm WARN chattrbox@1.0.0 No description
npm WARN chattrbox@1.0.0 No repository field.
```

留意`npm install`里的`--save-dev`选项。该选项告诉npm维护一份列表，记录应用依赖的所有第三方模块。这个列表存储在package.json文件中。如有必要，运行`npm install`命令（不带参数）即可安装列表里的全部依赖。如此一来，共享代码时就不必包含第三方模块了。

打开package.json文件，可以看到npm创建了`"devDependencies"`字段，其中有一条`nodemon`相关信息。

```
...
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "nodemon": "^1.9.1"
}
```

更新package.json，向`"scripts"`字段中添加一条新的命令：

```
...
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "node index.js",
  "dev": "nodemon index.js"
},
...
```

在终端运行`npm run dev`，重启node程序。注意，现在这条命令不是简单的`npm dev`了，它和`npm start`有所不同——因为npm会假定这种命令（比如`start`）一定存在，而自定义的npm

脚本则需要显式地强调你想要run这些脚本。

接下来你会看到nodemon已经在管理node程序了（如图15-9所示）。

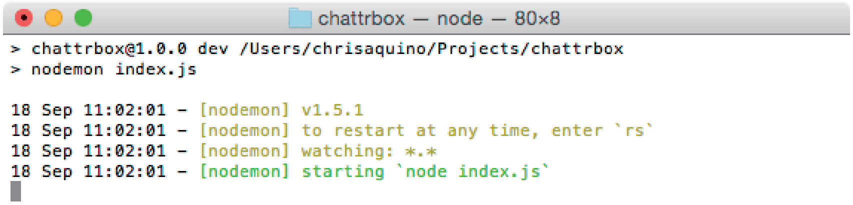


图15-9　通过npm run dev运行程序

在index.js中将“Hello, World”修改为“Hello, World!!”，保存修改。nodemon会发现并自动重启node程序（如图15-10所示）。

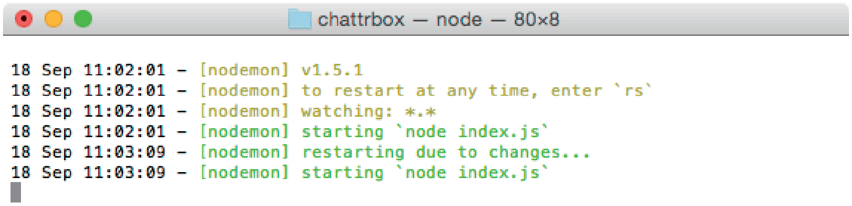


图15-10　代码改变时，nodemon重启程序

在接下来的几章继续使用node和npm的过程中，会不时引进新模块帮助我们完成任务。

15.4　用文件提供服务

在服务器端编写和运行JavaScript是一件很美妙的事情。大多数服务器还希望分发和处理文件内容。下一步就是让服务器从子目录中读取文件，并将其通过响应返回给浏览器——这有点像前几章的browser-sync。

在chattrbox项目目录下新建app文件夹，并在其中创建index.html文件，写入以下内容：

```
Hello, File!
```

该文件不必包含实际的HTML，只要写点能被读取的内容即可。现在的项目目录看起来如图15-11所示。

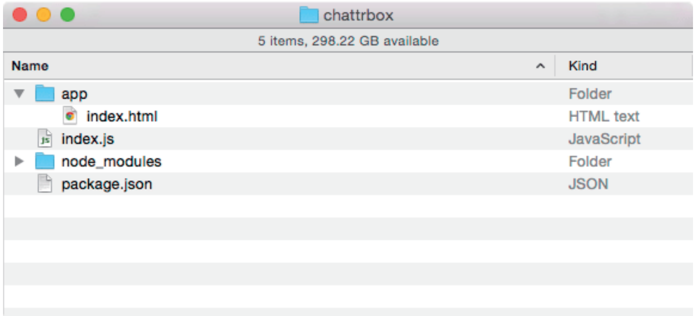


图15-11　Chattrbox项目结构

15.4.1　用fs模块读取文件

在index.js中引入Node.js文件系统模块`fs`，调用`fs`的`readFile`方法。

```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function (req, res) {
  console.log('Responding to a request.');
  res.end('<h1>Hello, World</h1>');
  fs.readFile('app/index.html', function (err, data) {
    res.end(data);
  });
});
server.listen(3000);
```

`readFile`方法接受一个文件名和一个回调函数作为参数。在回调函数中使用`res.end`返回文件内容，而不再是返回HTML文本。

注意，回调函数会在接受文件内容的同时接受一个`err`参数。这是Node.js的编程惯例，稍后会进行讨论。

`nodemon`会重启程序，因此直接打开浏览器并刷新页面即可。可以在浏览器中看到index.html文件中的内容：

```
Hello, File!
```

这是一个好的开始，但一个聊天应用要提供的可不止是一个HTML文件这么简单——这个HTML文件还要会请求其他的CSS或JavaScript文件。为了完成这些请求，你的node程序需要理解请求了哪个文件，在哪儿能找到请求的文件。接下来就实现这个功能。

15.4.2　处理请求URL

首先需要从请求对象中取得URL路径。如果路径只是'/'，最好返回index.html文件。这个命名从Web早期开始就是惯例了。

此外，需要返回请求对象要求的文件。

在index.js中更新你的回调函数，取得浏览器请求的文件路径。

```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function (req, res) {
  console.log('Responding to a request.');
  var url = req.url;

  var fileName = 'index.html';
  if (url.length > 1) {
    fileName = url.substring(1);
  }
  console.log(fileName);
  fs.readFile('app/index.html', function (err, data) {
    res.end(data);
  });
});
server.listen(3000);
```

从请求对象的`url`属性可以看出浏览器请求的到底是默认页面（index.html）还是别的文件。假如是其他文件，调用`url.substring(1)`去掉首字符，也就是'/'。

到现在为止，还只是将文件名打印到了控制台。

`nodemon`重启程序之后，在浏览器访问http://localhost:3000/woohoo或包括默认路径'/'在内的其他路径。终端结果如图15-12所示。

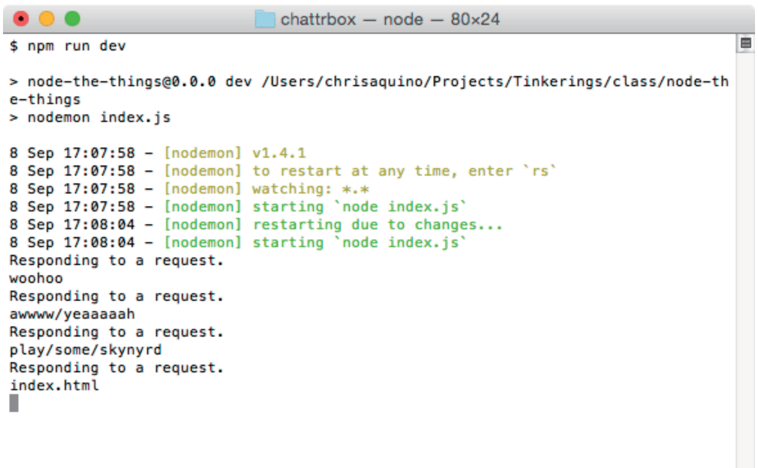


图15-12　打印请求的文件路径

（还记得在第2章中浏览器自动请求favicon.ico文件么？在终端也可以看到打印出的相应请求。）

现在该使用路径信息了。

15.4.3 使用path模块

刚才将`fileName`传给了`fs.readFile`，但最好使用path模块，它提供了可以处理和转换文件路径的公用函数。使用path模块的一个简单而重要的原因是，有些操作系统使用斜杠，而有些操作系统使用反斜杠，path模块会帮忙处理这些差异。

修改index.js，引入path模块，用它来查找请求的文件。

```
var http = require('http');
var fs = require('fs');
var path = require('path');

var server = http.createServer(function (req, res) {
  console.log('Responding to a request.');
  var url = req.url;

  var fileName = 'index.html';
  if (url.length > 1) {
    fileName = url.substring(1);
  }
  console.log(fileName);
  var filePath = path.resolve(__dirname, 'app', fileName);
  fs.readFile('app/index.html' filePath, function (err, data) {
    res.end(data);
  });
});
server.listen(3000);
```

在浏览器里输入几个测试的文件路径，确保程序的功能没变。默认路径应该返回index.html，不存在的路径（比如'/woohoo/'）应该不显示任何东西并且打印出文件名。

接下来，在app文件夹下创建test.html文件，写入下面的内容：

```
Hola, Node!
```

在浏览器中访问该资源，node程序会正常返回内容（如图15-13所示）。

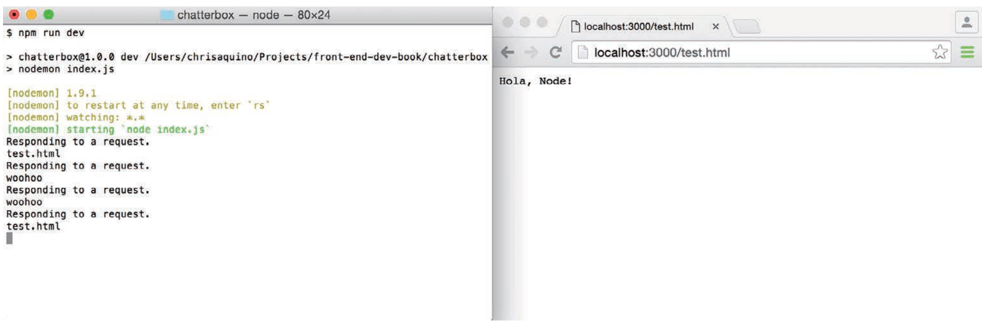


图15-13 取得test.html

现在的代码已经可以根据URL路径提供相应文件了。下一步则是将该功能函数抽象出来，成为独立模块。

15.4.4　创建自定义模块

前面的回调函数（至少）做了两件事情。首先分析出请求的文件是什么，然后读取文件内容并在响应中返回。为了让代码更加模块化且更好维护，需要将其中一项功能挪到单独的模块中。

在CoffeeRun里，可以在IIFE中声明模块，IIFE能给全局命名空间的属性赋值。但Node程序中的模块不太一样。现在还是在单独的文件中编写模块代码，但是不需要IIFE了。

在index.js的同级目录（不是app目录）下新建extract.js文件，添加一个`extractFilePath`函数用来查找对应的文件。（代码跟在index.js中写过的代码非常相似。）

```
var path = require('path');

var extractFilePath = function (url) {
  var filePath;
  var fileName = 'index.html';

  if (url.length > 1) {
    fileName = url.substring(1);
  }
  console.log('The fileName is: ' + fileName);

  filePath = path.resolve(__dirname, 'app', fileName);
  return filePath;
};
```

现在index.js中的很多代码都挪到了单独的函数`extractFilePath`中。接下来，让`extractFilePath`函数能被其他模块用`require`引入。要实现这一点，需将`extractFilePath`赋值给名为`module.exports`的全局变量。这是一个由Node提供的特殊变量，赋给它的值都能被其他模块引入，别的变量和函数则不能被其他模块访问。

```
...
  filePath = path.resolve(__dirname, 'app', fileName);
  return filePath;
};

module.exports = extractFilePath;
```

新增的一行代码告诉Node，当通过调用`require('./extract')`引入`extract`模块时，返回值是`extractFilePath`函数。下面就在index.js中引入它。

15.4.5　使用自定义模块

修改index.js，使用新的`extract`模块替代原来的功能代码。

```
var http = require('http');
var fs = require('fs');
var path = require('path');
var extract = require('./extract');

var server = http.createServer(function (req, res) {
  console.log('Responding to a request.');
```

```
  var url = req.url;

  var fileName = 'index.html';
  if (url.length > 1) {
    fileName = url.substring(1);
  }
  console.log(fileName);
  var filePath = path.resolve(__dirname, 'app', fileName);
  var filePath = extract(req.url);
  fs.readFile(filePath, function (err, data) {
    res.end(data);
  });
});
server.listen(3000);
```


用require函数引入自定义模块，并且将模块赋值给新的变量extract。然后，就能使用extract函数（等价于使用extractFilePath函数）了。

待nodemon重新启动程序后，用一些URL路径进行测试，确保默认的index.html和test.html还会加载，还要确保不存在的路径会返回空白页并且不会报错。

15.5　错误处理

现在还剩最后一项任务：当文件不存在时，最好返回错误码，而不是默默地假装一切正常。因此，当fs.readFile返回错误而不是文件的时候，我们需要感知错误。

在JavaScript中，经常将回调函数传给API方法。Node.js也是如此，回调函数一般都把错误作为第一个参数。因为把错误放在返回结果之前，所以不管是否处理它，至少能让人看到。

在index.js中检查是否有文件错误。如果有，则往响应中写入404错误码。

```
var http = require('http');
var fs = require('fs');
var extract = require('./extract');

var handleError = function (err, res) {
  res.writeHead(404);
  res.end();
};

var server = http.createServer(function (req, res) {
  console.log('Responding to a request.');
  var filePath = extract(req.url);
  fs.readFile(filePath, function (err, data) {
    if (err) {
      handleError(err, res);
      return;
    } else {
      res.end(data);
    }
  });
});
server.listen(3000);
```

保存修改。`nodemon`重启后，访问一个不存在的路径，比如http://localhost:3000/woohoo。在开发者工具中打开网络面板，就能看到错误码，如图15-14所示。

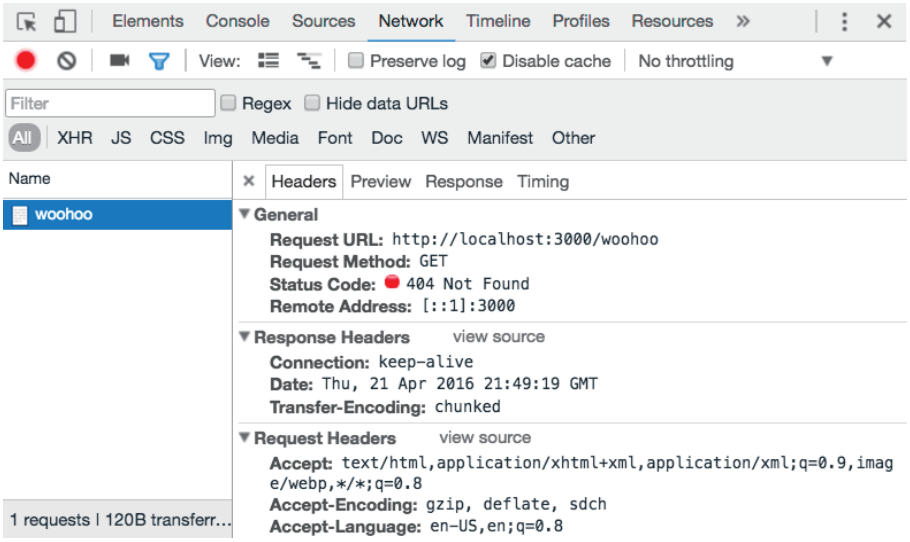


图15-14　网络面板里的404状态码

在回调函数中要做的第一件事情就是检查`err`参数是否不为`null`或`undefined`，并且进行处理。上面的例子先把错误信息传给函数`handleError`，然后`return`，就退出了匿名回调函数。

永远不要默默地丢弃错误。目前简单地返回404就够了，这也是`handleError`的做法。

“先报错，早返回”的模式是Node生态系统的最佳实践之一。Node的所有模块都遵循这一模式，大多数的开源模块也是如此。

到目前为止，我们已经使用了熟悉的模式（如回调函数），通过十几行JavaScript代码构建了一个可以运行的Web服务器。

Node提供了大量与网络、文件交互的模块，如在本项目里用到的`http`和`fs`。此外，Node的`require`和`module.exports`关键字让我们能轻松地模块化自己的代码。

接下来的3章会继续构建Chattrbox，同时还会开发相应的前端程序。

15.6　延展阅读：npm 模块注册

有大量可用的包能通过npm安装。在专门的模块注册网站，即www.npmjs.com上就能搜索和浏览这些包。

请阅读docs.npmjs.com上关于npm的文档。也许你还想自己创建模块给别人使用，那么可以看一下docs.npmjs.com/getting-started/creating-node-modules。

15.7　初级挑战：创建自定义错误页面

目前在访问一个不存在的页面路径时，服务器会返回空白页和一个404状态码。

那么现在的挑战是，创建一个专门的错误页面。当访问不存在的路径时，给用户展现这个页面，而不是一个状态码。

15.8　延展阅读：MIME 类型

你是否想过为什么计算机知道要用一个视频播放器打开一个电影文件，用一个文档查看器打开一个PDF文件？那是因为计算机维护了一个表格，用来保存文件类型和与文件类型相关联的程序。计算机根据文件的扩展名（如.html或者.pdf）来推断文件类型。

浏览器同样需要关联信息，这样它才能知道是将响应渲染成HTML，还是使用插件播放音乐，抑或是将文件下载到硬盘上。但HTTP响应没有这样的文件扩展名，因此服务器必须在响应里告诉浏览器响应信息的类型。

服务器通过在响应的`Content-Type`首部中指定**MIME类型**或者**媒体类型**来实现这一目标。举个例子，图15-15是在开发者工具的网络面板中观察到的www.bignerdranch.com的响应。

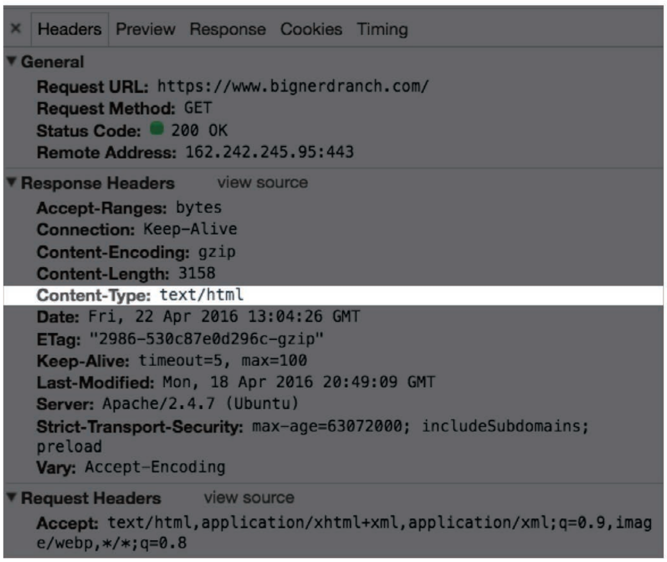


图15-15　观察www.bignerdranch.com的`Content-Type`首部

`Content-Type`首部被设置为`text/html`，即HTML的MIME类型。你也可以在项目里设置这样的首部信息。在Chattrbox项目里可以把代码改成这样：

```
...
var server = http.createServer(function (req, res) {
  console.log('Responding to a request.');
  var filePath = extract(req.url);
  fs.readFile(filePath, function (err, data) {
    if (err) {
      handleError(err, res);
      return;
    } else {
      res.setHeader('Content-Type', 'text/html');
      res.end(data);
    }
  });
});
server.listen(3000);
```

（注意，要在结束响应之前设置首部。）

如需获取更多关于MIME类型的信息，请访问en.wikipedia.org/wiki/Media_type。如需获取在Node程序里设置首部的相关信息，请访问nodejs.org/api/http.html#http_response_setheader_name_value。

15.9 中级挑战：动态提供 MIME 类型

请根据文件类型，给响应动态地提供MIME类型。为了更方便地实现这一功能，请用npm安装`mime`模块。关于`mime`模块的信息和文档可以在github.com/broofa/node-mime上找到。

往app文件夹下增加不同类型的文件（包括纯文本、PDF、音频文件、电影等），确保浏览器能够正确展示每种类型。

15.10 高级挑战：将错误处理放到单独的模块中

请将文件读取和错误处理的代码移到单独的模块中。

同时，要让模块可配置，这样就可以在引入该模块时指定静态HTML、CSS、JavaScript文件所在的基础目录。

第 16 章

使用WebSocket进行实时通信

如果用常规的GET和POST请求，每次与服务器进行数据交换时，浏览器都需要发起新的请求并等待响应。Ajax请求也是如此——虽然它不会导致页面重载，但是一样会产生网络流量，生成和处理每个请求和响应都需要产生一些开销。

相反，WebSocket提供了HTTP之上的双向通信协议。它创建一个单独的连接，而且保持连接打开，用来进行实时通信（如图16-1所示）。

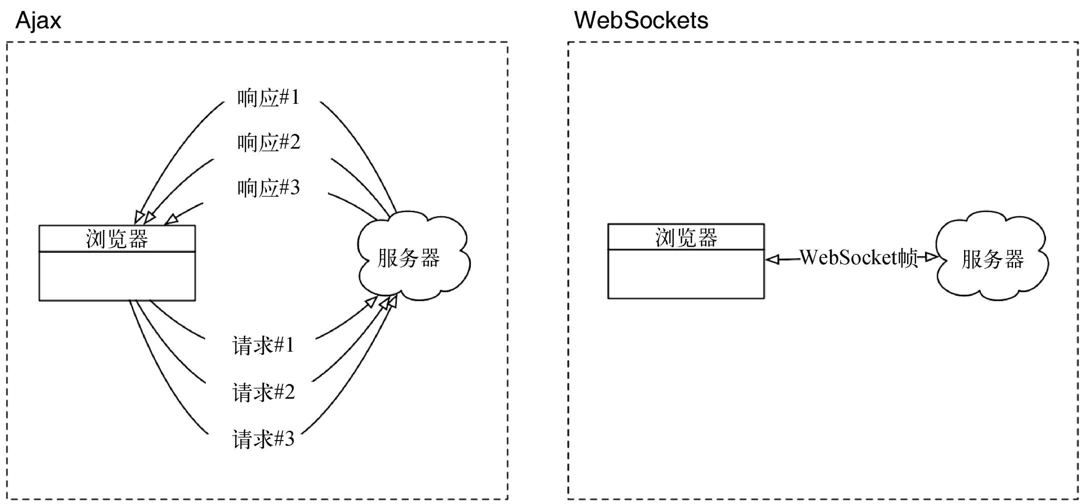


图16-1　多个Ajax请求 VS 一个单独的WebSocket连接

在Web应用中使用WebSocket不仅可以实现保存、加载远程数据，而且推送通知、文档协作编辑、实时聊天等都只能算是入门级别的功能。WebSocket使得服务器能够处理物联网上承载的事物（如智能灯光、智能锁、智能汽车等），但Ajax请求这种传统技术处理起如此密集的通信流量的效率极低。

本章会创建一个聊天应用的客户端和服务器。假如使用Ajax来构建这个应用，必须至少建立两个连接：一个用来请求新的消息，另一个用来发送消息。而用WebSocket则能在一个单独的连接上完成同样的事情。

在本章末尾，Chattrbox将能处理多个并发的聊天客户端，并将新消息发送到每个客户端（如图16-2所示）。

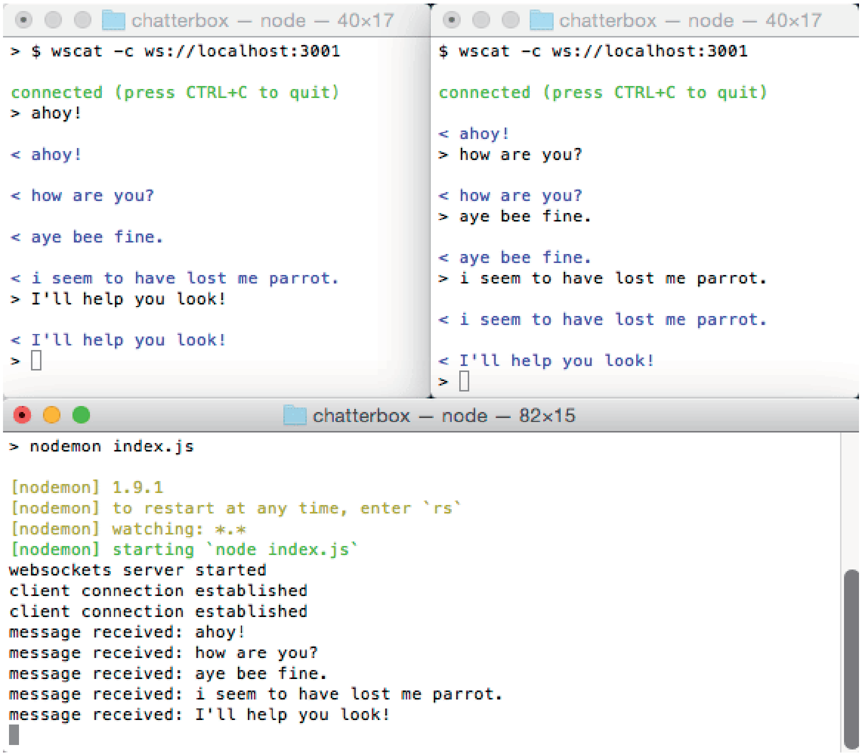


图16-2　海盗聊天室

16.1　配置 WebSocket

为了在Chattrbox中使用WebSocket，需要执行以下步骤。

(1) 安装`ws`模块。

(2) 创建WebSocket服务器。

(3) 为服务器添加聊天功能。

(4) 向新用户发送历史聊天消息。

从第一步开始。

Node.js中的`http`模块提供了启动HTTP服务器的简单方式，方便浏览器和服务器进行通信。

和http相似，通过WebSocket，ws模块向Node.js程序提供了简单的通信方式。很多模块都实现了WebSocket，但是ws是标准实现，性能很好。

首先在Chattrbox目录下安装WebSocket模块ws。（如果看见npm发出警告，提示项目缺失了描述或者仓库信息，请不要惊慌。）

```
npm install --save ws
```

下一步，在Chattrbox根目录下新建websockets-server.js文件。添加代码引入ws模块，开始监听：


```
var WebSocket = require('ws');
var WebSocketServer = WebSocket.Server;
var port = 3001;
var ws = new WebSocketServer({
    port: port
});

console.log('websockets server started');
```

使用require声明导入ws模块。该模块包含了一个Server属性，使用该属性可以创建一个可用的WebSocket服务器。

这一功能的核心代码是var ws = new WebSocketServer(/*...*/);。运行这段代码会创建WebSocket服务器，并绑定指定的端口号（这里是3001）。

跟extract.js中的模块不同，这里不需要module.exports赋值操作。引入websockets-server.js模块时，其中的代码就会运行。该模块会处理所有关于WebSocket的初始化和事件处理。

创建好WebSocket服务器之后的第一件事情就是处理连接。在websockets-server.js中，为WebSocket服务器中所有的连接事件创建一个回调函数：

```
Var WebSocket = require('ws');
var WebSocketServer = WebSocket.Server;
var port = 3001;
var ws = new WebSocketServer({
    port: port
});

console.log('websockets server started');

ws.on('connection', function (socket) {
  console.log('client connection established');
});
```

事件处理语法与jQuery类似，很多JavaScript库（包括Node、浏览器端）都使用了该模式。

事件处理回调函数唯一接受的参数叫作socket。当一个客户端与WebSocket服务器建立连接时，就能通过这个socket对象获取这个连接。

在写聊天应用的服务端代码之前，需要配置服务端程序，使其能够重复任意接受到的消息。这种服务器通常叫做**回声服务器**。

给客户端连接上产生的任意message事件注册一个回调函数，将“回声”功能添加到websockets-server.js中。

```
...
console.log('websockets server started');

ws.on('connection', function (socket) {
  console.log('client connection established');

  socket.on('message', function (data) {
    console.log('message received: ' + data);
    socket.send(data);
  });
});
```

将事件处理程序直接注册到socket对象上。message事件回调函数会接受客户端发送的任何信息。现在，聊天程序只是在相同的socket连接上将收到的信息send回去。

马上就能看到实际运行效果。

可以直接在命令行执行node websockets-server.js启动WebSocket服务器，但是把它放到index.js中执行也不麻烦。而且后者还有一个好处：不管对websockets-server.js还是index.js进行修改，都可以利用nodemon自动重载代码。

在index.js顶部添加一个require声明，引入websockets-server模块。

```
var http = require('http');
var fs = require('fs');
var extract = require('./extract');
var wss = require('./websockets-server');
...
```

保存文件，nodemon会重载代码。一切就绪，只等检测效果。

16.2　测试 WebSocket 服务器

一个简单的测试方法是使用wscat模块。wscat工具可以用来连接WebSocket服务器并与之通信。该模块提供了命令行程序，可以当作一个聊天应用的客户端。

新打开一个终端窗口，全局安装wscat。可能需要使用管理员权限来执行安装命令。（可以参考第1章，复习相关知识。）

```
npm install -g wscat
```

安装完wscat，就可以连接到WebSocket服务器了。

在第二个终端窗口中运行wscat -c ws://localhost:3001，随后在第二个终端窗口就能看到消息connected (press CTRL+C to quit)，在第一个终端窗口则能看到'client connection established'。

在第二个终端窗口的光标处输入一些文字。每次输完文字按下回车键，输入的文字都会被WebSocket服务器返回。如图16-3所示。

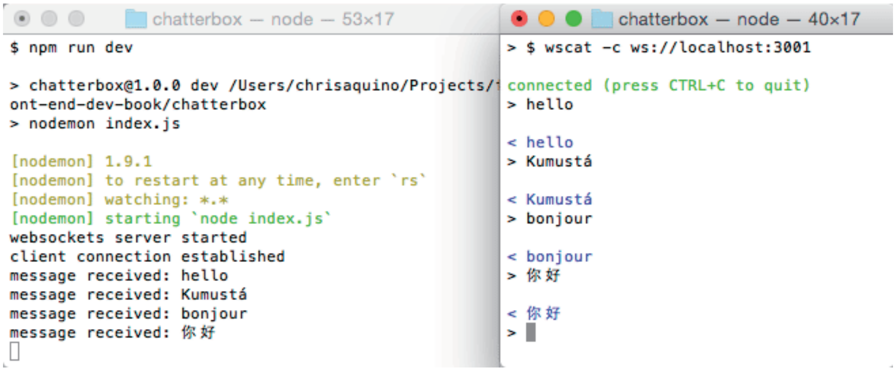


图16-3　使用wscat测试服务器

现在已经能通过WebSocket与服务器进行通信。接下来就要添加一些真正有用的功能，让Chattrbox成为名副其实的聊天系统。

16.3　创建聊天服务器的功能

启动WebSocket服务器之后，就能够构建聊天服务器了。聊天服务器应具备以下功能。

- ❑ 记录发送到服务器的所有消息。
- ❑ 向新加入聊天室的用户广播历史消息。
- ❑ 向所有客户端广播新消息。

将用户发送的消息记录下来才能向新用户发送历史消息，所以要先实现记录消息的功能。在websockets-server.js中创建一个数组，用来记录消息。

```
var WebSocket = require('ws');
var WebSocketServer = WebSocket.Server;
var port = 3001;
var ws = new WebSocketServer({
    port: port
});
var messages = [];

console.log('websockets server started');
...
```

如果要创建更健壮的聊天系统，应该将消息存储在数据库中。不过目前用一个简单的数组就可以了。

接下来，当收到新消息时，调用`messages.push(data)`将新消息添加到数组中。

```
...
ws.on('connection', function (socket) {
  console.log('client connection established');

  socket.on('message', function (data) {
```

```
    console.log('message received: ' + data);
    messages.push(data);
    socket.send(data);
  });
});
```

上面代码中的数组能够将聊天服务器收到的消息保存起来。

下一步就是让新用户看到所有的历史消息。修改websockets-server.js中的连接事件处理程序，向每个新到达的连接发送所有的历史消息。

```
...
ws.on('connection', function (socket) {
  console.log('client connection established');

  messages.forEach(function (msg) {
    socket.send(msg);
  });

  socket.on('message', function (data) {
    console.log('message received: ' + data);
    messages.push(data);
    socket.send(data);
  });
});
```

当建立了一个连接之后，服务器遍历所有消息，把每条消息发送给新的连接。

最后一项任务是：当接受到新消息时，将新消息发送给所有用户。WebSocket记录了所有已连接用户。在websockets-server.js中使用这一机制，将收到的消息再次广播。

```
...
ws.on('connection', function (socket) {
  console.log('client connection established');

  messages.forEach(function (msg) {
    socket.send(msg);
  });

  socket.on('message', function (data) {
    console.log('message received: ' + data);
    messages.push(data);
    ws.clients.forEach(function (clientSocket) {
      clientSocket.send(data);
    });

    socket.send(data);
  });
});
```

`ws`对象用`clients`属性记录了所有的连接。该属性是一个数组，可以对其进行迭代。在迭代的回调函数中，只需要`send`消息数据。

最后，因为在迭代所有的socket连接时，已经将消息发送到了当前的socket连接，因此没必要再调用`socket.send(data)`了。将其删除，免得重复发送消息。

16.4　第一次聊天！

现在来测试新功能。先确保nodemon已经重新加载了代码。（如有必要，可以按Control + C手动停止nodemon并输入npm run dev重启。）

打开第三个终端窗口，运行命令wscat -c http://localhost:3001。（需要一个终端窗口运行nodemon，另外两个窗口运行wscat。）在连接到服务器的两个窗口中输入一些聊天消息。

自己跟自己聊一会儿后，打开第四个终端窗口，运行wscat -c http://localhost:3001。这一个聊天客户端应该能收到所有的历史消息。

不出意外的话，应该能看到如图16-4所示的情形。

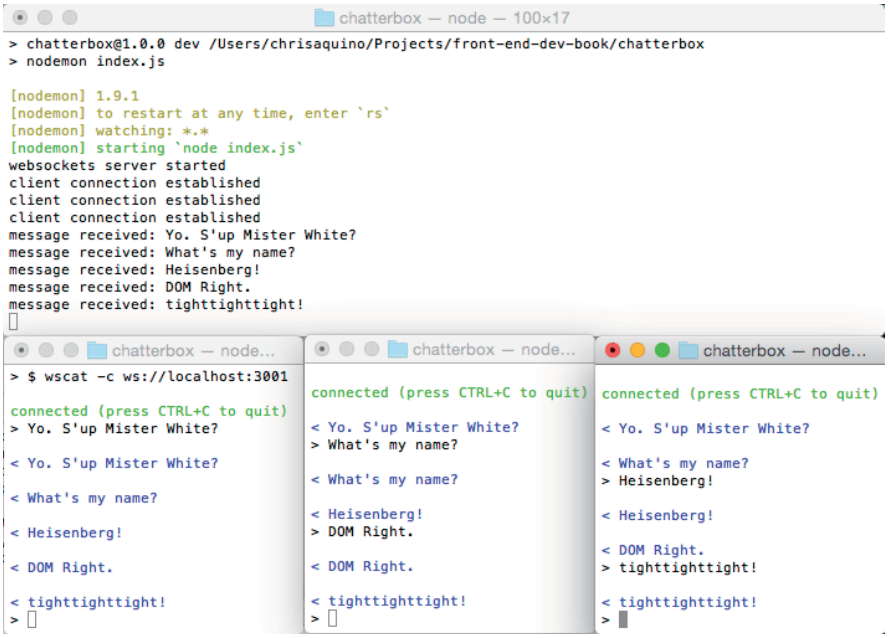


图16-4　与几个朋友聊天

恭喜你！已经用WebSocket完成了一个功能齐备的聊天服务器——而且仅仅用了二十几行JavaScript代码。

16.5　延展阅读：WebSocket 库 socket.io

ws模块是WebSocket的不错实现。但必须承认，它缺少一些方法。举个例子，WebSocket连接有时候会掉线，但是ws模块没有提供自动重连的方法。

还有一个问题，ws只存在于Node.js的世界里，只能在服务端使用。在客户端的JavaScript中，还得学习并使用另一个完全不同的库，哪怕它们实现的核心功能一模一样。

另外，在客户端还有其他问题：假如浏览器版本很低，不支持WebSocket怎么办？因此，还需要一个降级方案。

`socket.io`（socket.io）为这些问题提供了解决办法——它为浏览器提供了向后兼容的降级方案，包括一个Flash的实现方案。此外，它还被移植到了很多其他平台，包括iOS和Android。

16.6　延展阅读：WebSocket 服务

假如你对提供实时平台的服务感兴趣，可以试一下`firebase`（www.firebase.com）。如果说`socket.io`降低了编写服务端代码的难度，`firebase`则更进一步——它提供了一整套服务，包括让客户端共享和同步数据的机制。`firebase`为Web、iOS和Android等平台均提供了解决方案。

16.7　初级挑战：我重复了我的消息吗？

更新`message`事件处理程序，使收到的每条消息向每个用户发送两次。

用`wscat`进行测试，确认每条消息都重复了。

为了实现有趣的效果，每次收到新消息时，增加一次重复次数。

16.8　中级挑战：Speakeasy

在20世纪20年代的美国，酒的生产和销售都是非法的，于是人们创造出speakeasy——售卖酒精饮料的秘密酒吧。这些酒吧要求顾客报上密码，方能让其进入。

给聊天程序也创建一个speakeasy版本吧（但是咱不卖酒），将所有的消息都隐藏起来，直到用户输入密码。（出于历史原因，Swordfish这个密码不错。）

当用户输入了密码，将所有历史消息发送给用户，并且允许他们看到新消息。

16.9　高级挑战：聊天机器人

前面使用`WebSocket.Server`属性创建了聊天服务器。你也可以使用`WebSocket`作为构造函数，用程序创建聊天客户端。

下面是示例代码：

```
var chatClient = new WebSocket('http://localhost:3001');
```

在github.com/websockets/ws上的文档中有一个简单的例子，可以发送和接收文本数据。

创建一个聊天机器人，让它能自动连接到聊天服务器。它能跟每个新用户打招呼，其他时候保持沉默，直到有用户直接与它对话。举个例子，假如你的聊天机器人能响应Jinx这个名字，那么你输入Jinx, put Max in space的话，聊天机器人就会根据你的输入进行回答。（回答的内容取决于你。）

确保聊天机器人的代码在一个单独的模块中，而不要将其直接构建到聊天服务器代码中。

第 17 章

借助Babel使用ES6

JavaScript语言诞生于1994年，在1999年进行了一些更新，但是从1999年到2009年就一直没有更改过。在2009年引入了一系列较小的修改后，我们所熟知的ES5版本，或者说是标准第五版便诞生了。

2015年，标准第六版增加了很多语言改进，其中许多新的语言特性受到了Ruby和Python等语言的影响。严格来讲，第六版被命名为ES2015，但是更普遍的叫法是ES6。

ES6在谷歌的Chrome浏览器、Mozilla的火狐浏览器，以及微软的Edge浏览器上都得到了非常好的支持。这些都是**长青**的浏览器，即它们会自动更新，无须用户手动下载或安装最新版本。随着谷歌、Mozilla以及微软的浏览器对ES6的兼容性越来越好，开发者很快就能使用这些语言上的改进了。

但是那些非长青的浏览器，以及大多数手机浏览器对ES6的支持非常差。图17-1是最新版本的桌面、手机浏览器对ES6特性的支持百分比。（图中，IE = Internet Explorer、FF = Mozilla Firefox、CH = Google Chrome、SF = Safari、KQ = Konqueror、AN = Android。）

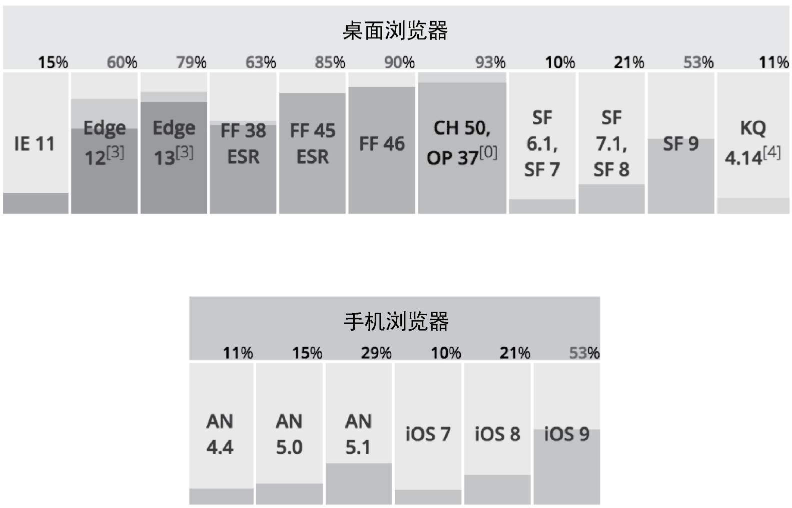


图17-1　2016年春，ES6特性的支持情况

如果要看各大浏览器对ES6更新更详细的支持情况，请访问kangax.github.io/compat-table/es6/查看最新信息，创建者Juriy Zaytsev一直在更新表格数据。

老旧的浏览器支持的特性很少，但是这无法抵挡我们对ES6的钟爱。ES6非常棒，你值得拥有，不必等到所有浏览器都支持它的时候才开始使用。

本章会开始开发Chattrbox的用户界面，在开发过程中会用到很多ES6特性。为了让应用能在所有浏览器中运行，我们将使用开源工具Babel处理兼容性问题。

在开始之前，还需要处理一件事，以便更专注地学习ES6并使用Babel。Chattrbox项目的index.html和stylesheets/styles.css文件放在了www.bignerdranch.com/downloads/front-end-dev-resou rces.zip上。请下载.zip文件，提取其中的内容（包括整个stylesheets/文件夹），将内容复制到chattrbox/app路径下。（index.html会替换项目下面已有的index.html文件。）

另外，提醒一下：在编写本章代码的过程中，可能会在控制台看到关于CSS文件的MIME类型的警告。大可放心，忽略这个警告吧。

继续往前推进！待到本章结束，Chattrbox将能够通过WebSocket和聊天服务器通信。

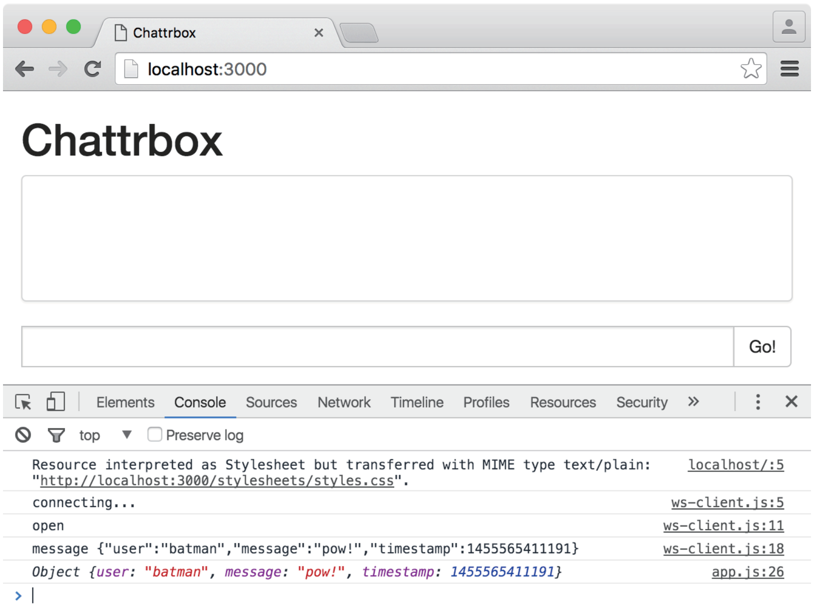


图17-2　本章结束时的Chattrbox

17.1　编译 JavaScript 的工具

Babel是一个编译器，它负责将ES6语法翻译成等价的ES5代码，这样就能够在浏览器的JavaScript引擎下运行了。

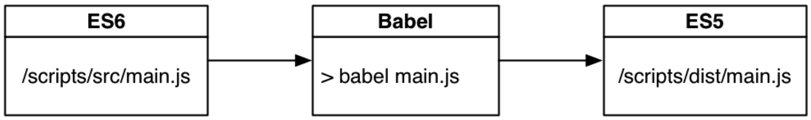


图17-3 根据ES6文件构建ES5代码

为了更高效地使用Babel，还需要安装几个npm模块来运行自动构建流程。用Babel将ES6编译成ES5，用Browserify将模块打包到一个单独的文件，用Babelify让两者（Babel和Browserify）协同工作。另外还要用Watchify实时监听代码变化，触发构建流程（如图17-4所示）。

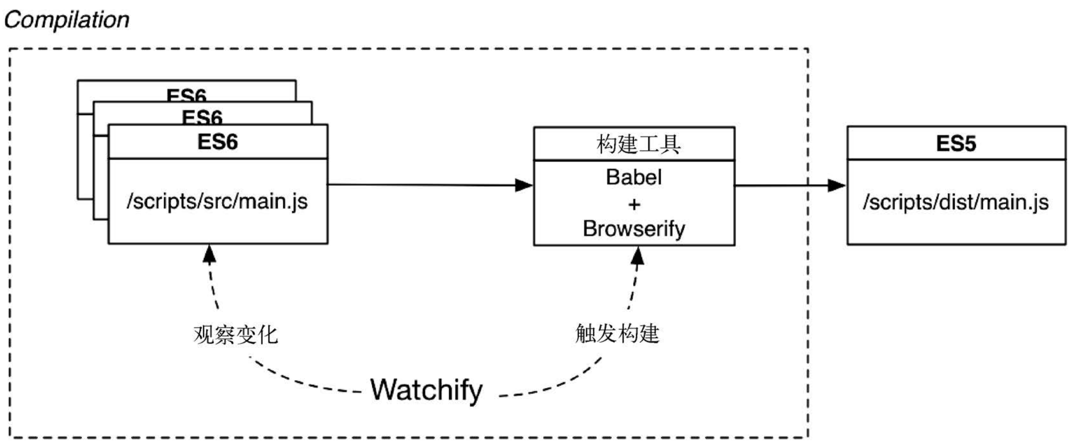


图17-4 编译流程

首先需要安装Babel。Babel有一些不同的用法，需要根据具体需求来决定。在Chattrbox项目里，需要用两种方式进行编译：命令行或者程序。babel-cli和babel-core这两个工具分别能够满足这两个需求。此外，还需要安装Babel配置，用来编译ES6标准，这个配置叫做babel-preset-es2015。

在chattrbox目录运行下面的npm命令，安装合适的Babel工具。（参考第1章，复习如何使用管理员权限运行`npm install -g`。）

```
npm install -g babel-cli
npm install --save-dev babel-core
npm install --save-dev babel-preset-es2015
```

接下来用刚才安装的es2015预设配置来配置Babel。在chattrbox根目录下创建一个名为.babelrc的文件，写入下面的配置信息：

```
{
  "presets": [
    "es2015"
  ],
  "plugins": []
}
```

最后，将Babelify、Browserify、Watchify安装到chattrbox/node_modules/目录下：

```
npm install --save-dev browserify babelify watchify
```

在启动Babel后不久就会用到这三个工具。

17.2　Chattrbox 客户端应用程序

前面两章已经将Chattrbox服务端构建好了，服务端能够返回静态文件，并且通过WebSocket通信。客户端应用则要通过WebSocket与服务端相互收发消息。客户端应用将为每条消息定义一个格式。用户可以看到消息列表，还能通过在表单输入文字创建新消息。

这些功能将由以下3个模块处理。

- ❑ `ws-client`模块为客户端程序管理WebSocket通信。
- ❑ `dom`模块向UI展示数据，并处理表单提交。
- ❑ `app`模块定义消息结构，在`ws-client`和`dom`之间传递消息。

图17-5展示了三个模块之间的关系。

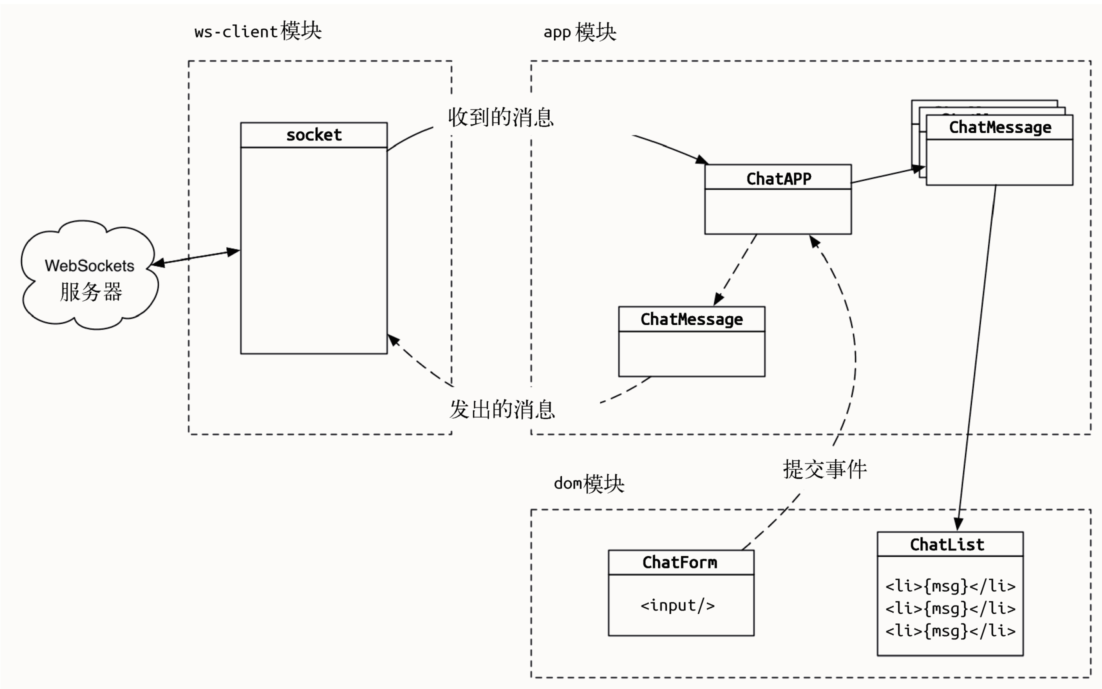


图17-5　Chattrbox应用程序模块

在chattrbox/app文件夹下创建scripts、scripts/dist以及scripts/src子文件夹，如图17-6所示。

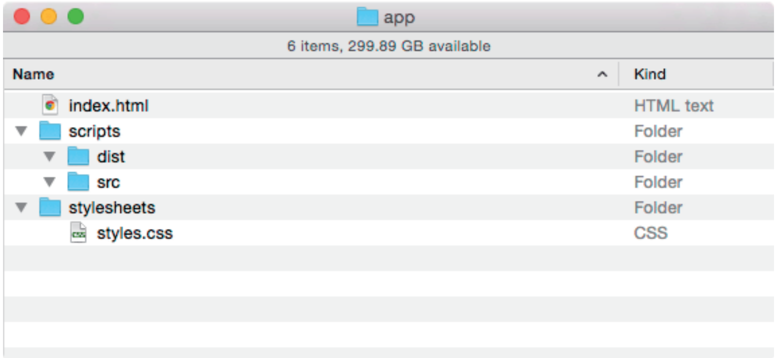


图17-6　chattrbox/app文件夹结构

现在在scripts/src目录下创建如下4个JavaScript文件：

- ❑ app.js
- ❑ dom.js
- ❑ main.js
- ❑ ws-client.js

现在的文件结构变成了如图17-7所示的样子。

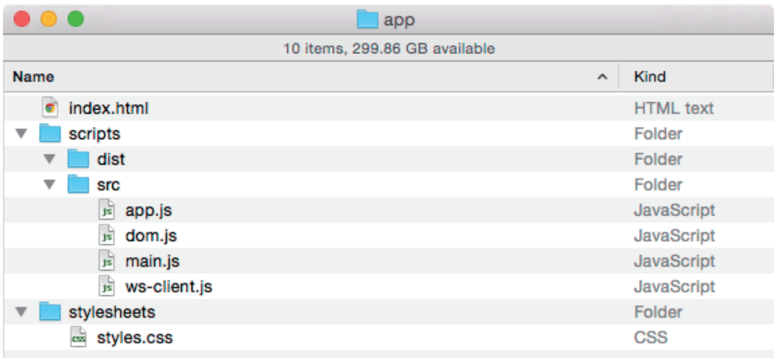


图17-7　chattrbox/app

app.js、dom.js以及ws-client.js对应了图17-5里的3个模块。main.js文件包含了应用的初始化代码。

17.3　迈出 Babel 的第一步

前面已经安装了项目所需的工具，项目文件也已创建完毕。现在开始ES6之旅。

目前暂时只在命令行使用Babel。稍后会将命令添加到npm脚本中，实现自动编译。这样，在

使用ES6特性时便可以专注于新语法，而不必在终端执行额外的命令。

class语法

构建Chattrbox客户端程序要用到的第一个ES6特性是class关键字。请牢记，ES6的class关键字跟其他编程语言里的类并不完全一样，ES6的类只是为构造函数和prototype方法提供了一种简写的语法。

打开app.js，定义一个新的类，命名为ChatApp。

```
class ChatApp {
}
```

在本章中，ChatApp没有太多功能，不过最后应用里的大部分逻辑都会在ChatApp中实现。

现在这个类的定义还是空的。添加一个constructor方法，写入一个console.log声明：

```
class ChatApp {
  constructor() {
    console.log('Hello ES6!');
  }
}
```

每当实例化一个类时，都会执行constructor方法。通常，构造函数会给实例的属性赋值。

接着，在app.js中的ChatApp类声明之后创建一个ChatApp实例。

```
class ChatApp {
  constructor() {
    console.log('Hello ES6!');
  }
}
new ChatApp();
```

试运行一下代码。打开第二个终端窗口，切换到Chattrbox根目录下，也就是package.json、index.js以及app/所在的目录。在这个窗口运行构建工具，在之前那个窗口运行服务端代码。

为了测试代码，用Babel编译app/scripts/src/app.js，将结果输出到app/scripts/dist/main.js中：

```
babel app/scripts/src/app.js -o app/scripts/dist/main.js
```

如果在终端什么也没发生，这就对了，就是这样的。除非有错误，否则Babel不会在终端输出任何内容（如图17-8所示）。

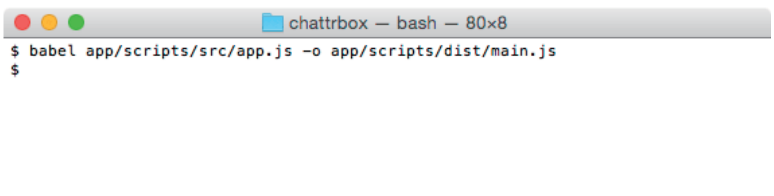


图17-8　Babel会默默地执行

确保让Node服务器在另一个终端里面运行（使用`npm run dev`），然后打开浏览器，输入http://localhost:3000。现在应该能看到结果了（如图17-9所示）。

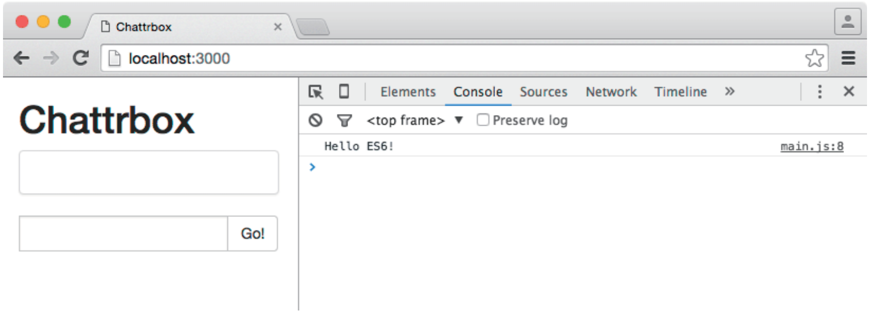


图17-9　Hello ES6!

在app/index.html中加载了由app.js生成的main.js。因为app.js创建了一个新的`ChatApp`实例，所以会运行`ChatApp`中的构造函数，并打印出Hello ES6!。

现在已经用Babel处理过单个JavaScript文件了，接下来就要处理多个模块（文件）了。

17.4　使用 Browserify 打包模块

ES5没有内置的模块系统。在前面构建CoffeeRun时，曾使用过一个变通的办法来写模块代码，但这需要修改一个全局变量。

ES6提供了真正的模块，就像其他语言的模块一样。Babel能够理解ES6模块语法，但是没法将其转换成等价的ES5代码，所以就需要使用Browserify了。

图17-10展示了Browserify和Babel搭配使用的方式。

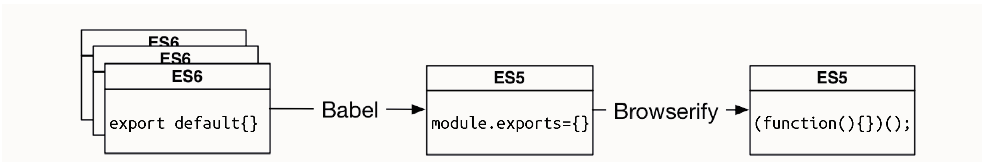


图17-10　使用Babel和Browserify将ES6模块转换成ES5模块

默认情况下，Babel将ES6模块语法转换成等价的Node.js风格的`require`和`module.exports`语法。接着Browserify将Node.js模块代码转换成ES5可以识别的函数。

打开package.json为Browserify添加一段配置：

```
...
  "scripts": {
    "test": "echo \"Error: no test specified\" &&
     exit 1",
    "start": "node index.js",
```

```
    "dev": "nodemon index.js",
  },
  "browserify": {
    "transform": [
      ["babelify", {"presets": ["es2015"], "sourceMap": true}]
    ]
  },
...
```

这段代码告诉Browserify将Babelify当作一个插件。它给Babelify传递了两个选项：第一个激活了ES2015编译器选项；另一个启用了sourceMap选项，这样有助于调试。在接下来构建Chattrbox的过程中，会介绍怎样使用source maps进行调试。

跟nodemon一样，最好为通用的Browserify任务编写一些脚本。打开package.json在"scripts"字段写入以下代码。（记得在"dev": "nodemon index.js"结尾加上逗号。）

```
...
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node index.js",
    "dev": "nodemon index.js",
    "build": "browserify -d app/scripts/src/main.js -o app/scripts/dist/main.js",
    "watch": "watchify -v -d app/scripts/src/main.js -o app/scripts/dist/main.js"
  },
  "browserify": {
    "transform": [
      ["babelify", {"presets": ["es2015"], "sourceMap": true}]
    ]
  },
...
```

第一个脚本build直接使用browserify命令。第二个脚本watch则在代码改变时使用watchify重新运行browserify（跟nodemon类似的功能）。

现在开始使用ES6模块系统。在ES6模块中，必须明确地导出想让别人使用的模块代码。修改app.js，导出ChatApp类，而不是简单地创建一个实例。

```
class ChatApp {
  constructor() {
    console.log('Hello ES6!');
  }
}
new ChatApp();
export default ChatApp;
```

这段代码指定ChatApp就是app模块里面可用的默认值。其他模块可能会导出多个值。如果只需要导出一个值，最好使用export default。

在main.js中导入ChatApp类，创建一个实例。

```
import ChatApp from './app';
new ChatApp();
```

main.js将app.js导出的ChatApp类导入进来。导入之后，创建一个ChatApp类的实例。

注意，在main.js中导入的类名是否叫`ChatApp`并不重要，因为`ChatApp`是app.js默认的导出值。比如，写成`import MyChatApp from'./app'`就会将默认的导出值赋给当前作用域下的`MyChatApp`变量名。只不过在导入时命名为`ChatApp`是最佳实践，因为它在app.js中的名字就是如此。

执行构建操作

现在打开终端，运行构建脚本：

```
npm run build
```

npm会运行`build`命令，该命令将调用`browserify`。运行每个命令时，都会显示当前正在执行的操作。但Browserify不会打印任何信息，除非遇到错误（如图17-11所示）。

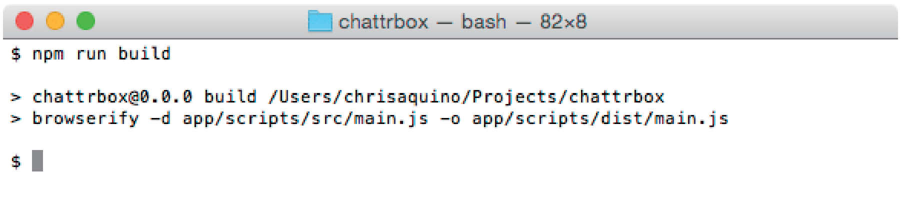


图17-11　通过`npm run build`运行Browserify

Browserify成功运行后，就会将app/dist/文件夹下Babel编译生成的main.js（就是前面手动编译的结果）打包。

重新加载浏览器就能看到输出。这里并没有新增功能，只修改了`ChatApp`构造函数的调用位置，因此在控制台看到的消息与之前一样（如图17-12所示）。

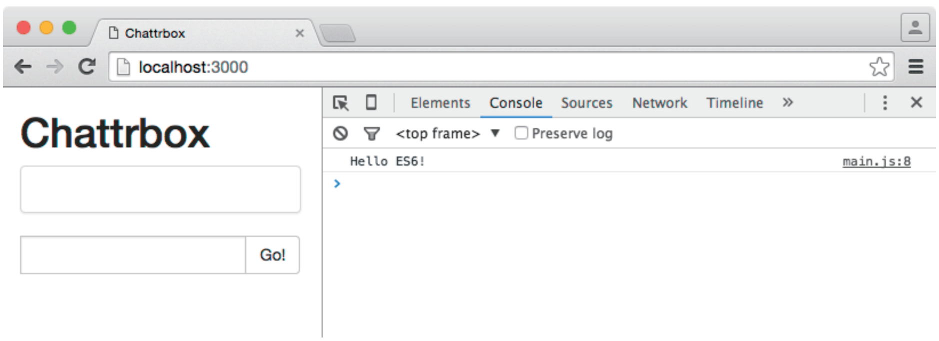


图17-12　再次Hello！

下一步是使用Watchify。就像用`nodemon`运行Node.js服务器一样，Watchify可以用来运行Browserify编译程序——只要修改了源文件，它就会自动触发重新编译。

启动Watchify。只要修改代码，就会开始编译：

```
npm run watch
```

Watchify会在控制台显示编译状态（如图17-13所示）。

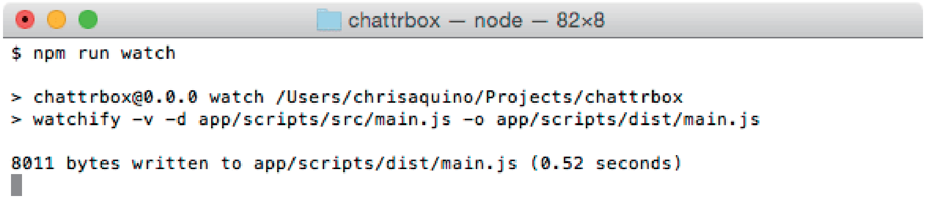


图17-13　通过`npm run watch`运行Watchify

Watchify比Browserify“健谈”多了——它每次运行Browserify时，都会显示向文件里写入了多少字节。虽然也没太大用处，但是输出发生改变时，你都能得到通知。开着终端让Watchify保持运行，继续开发Chattrbox。（之前还有一个终端运行着服务端程序。）

17.5　新增 ChatMessage 类

虽然用两个终端聊天很好玩（还能让你在咖啡馆看起来酷酷的），不过是时候升级应用，让它能在浏览器之间发送消息了。接下来新增一个辅助类，用于构造和格式化消息数据。

每条消息需要记录三类信息：消息内容、是谁发送的和什么时候发送的。

JavaScript对象表示法（JavaScript Object Notation）——更常见的说法是JSON（发音同Jason，源自JSON发明者Douglas Crockford）——是一个轻量级的数据交换格式，你在package.json文件中已经用到了它。这种格式具备可读性，独立于语言，而且很适合Chattrbox的数据交换。

这里有一个简单的JSON格式的消息：

```
{
  "message": "I'm Batman",
  "user": "batman",
  "timestamp": 614653200000
}
```

Chattrbox的消息数据有两个来源：一个是客户端，由用户填写的表单产生；另一个是服务端，经由一个WebSocket连接将消息发送到其他客户端。

对于表单产生的消息，需要先给消息体增加用户名和时间戳，然后才能将其发给服务端。对于服务端产生的数据，这三类信息都会包含在内。如何处理这种差异呢？有很多解决办法，下面简单介绍几个，其中一些还利用了便捷的ES6特性。

在app.js里新建一个`class`来表示每条聊天消息。

```
class ChatApp {
  constructor() {
    console.log('Hello ES6!');
  }
}
```

```
class ChatMessage {
  constructor(data) {
  }
}

export default ChatApp;
```

第一种方式就是使用简单的构造函数接收消息内容、用户名以及时间戳。（下面只是个例子，请不要据此修改你的项目代码。）

```
...
class ChatMessage {
  constructor(message, user, timestamp) {
    this.message = message;
    this.user = user || 'batman';
    this.timestamp = timestamp || (new Date()).getTime();
  }
}
...
```

这种模式在前面已经出现很多次了。将参数值赋给实例的属性，使用`||`操作符为用户名和时间戳提供一个默认值。

这种方式没问题，不过ES6提供的**默认参数值**能以更简洁的方式实现相同的模式。

```
...
class ChatMessage {
  constructor(message, user='batman', timestamp=(new Date()).getTime()) {
    this.message = message;
    this.user = user;
    this.timestamp = timestamp;
  }
}
...
```

该语法可以很清楚地表示哪些参数必须传，哪些参数可选。在上面的代码里，只有`message`参数是必需的，其余参数都有默认值。

上面的构造函数可以处理服务端发送的消息或用户通过表单创建的消息，但要求调用者清楚参数的顺序。一旦某些函数、方法需要3个或3个以上参数时，这种传参方式就不够灵活了。

还有一种方式是将一个单独的对象作为参数，使用键值对来指定消息内容、用户 名和时间戳。这种方式可以使用**解构赋值语法**（destructuring assignment syntax）来实现。

```
...
class ChatMessage {
  constructor({message: m, user: u, timestamp: t}) {
    this.message = m;
    this.user = u;
    this.timestamp = t;
  }
}
...
```

17

解构看上去可能有点奇怪，下面就是它的使用方法。像下面这样调用构造函数：

```
new ChatMessage({message: 'hello from the outside',
                 user: 'adele25@bignerdranch.com', timestamp=1462399523859});
```

解构语法在参数中查找message键，发现有值'hello from the outside'，于是赋值给一个新的局部变量m，然后就可以在构造函数体中使用该变量了。对于username和timestamp属性也是如此。

但是使用上面这种方式就不能利用默认参数的便利性了。万幸，可以将默认参数和解构赋值结合起来，所以app.js里构造函数的最终版长下面这样：

```
...
  class ChatMessage {
  constructor(data){
    message: m,
    user: u='batman',
    timestamp: t=(new Date()).getTime()
}) {
    this.message = m;
    this.user = u;
    this.timestamp = t;
  }
}
...
```

这一版代码把传给构造函数的对象里的值提取出来。任何没有赋值的参数都有默认值。

尽管默认参数只能存在于函数（或者构造函数）定义里，解构却可以在赋值操作时使用。上面的构造函数也可以写成下面这样：

```
...
class ChatMessage {
  constructor(data) {
    var {message: m, user: u='batman', timestamp: t=(new Date()).getTime()} = data;
    this.message = m;
    this.user = u;
    this.timestamp = t;
  }
}
...
```

好，题外话就说完了。现在回到构建Chattrbox上。

ChatMessage类将所有重要信息存储为属性，但是实例还继承了ChatMessage的方法和其他信息，这使得ChatMessage的实例不适合通过WebSocket发送。因此，需要一个信息的简化版。

在app.js里添加一个序列化（serialize）方法，用于将ChatMessage里的属性转化成一个简单的JavaScript对象。

```
...
class ChatMessage {
  constructor({
    message: m,
    user: u='batman',
```

```
    timestamp: t=(new Date()).getTime()
  }) {
    this.message = m;
    this.user = u;
    this.timestamp = t;
  }
  serialize() {
    return {
      user: this.user,
      message: this.message,
      timestamp: this.timestamp
    };
  }
}

export default ChatApp;
```

ChatMessage类现在已经可用了，下面来开发Chattrbox的下一个模块。

17.6　创建 ws-client 模块

ws-client.js模块负责与Node WebSocket服务器通信。

它有4项职责。

- 连接到服务器。
- 在初次建立连接时执行初始化配置。
- 将到达的消息发给相应的处理程序。
- 向外发送消息。

看一下这些职责是如何关联到其他组件的（如图17-14所示）。

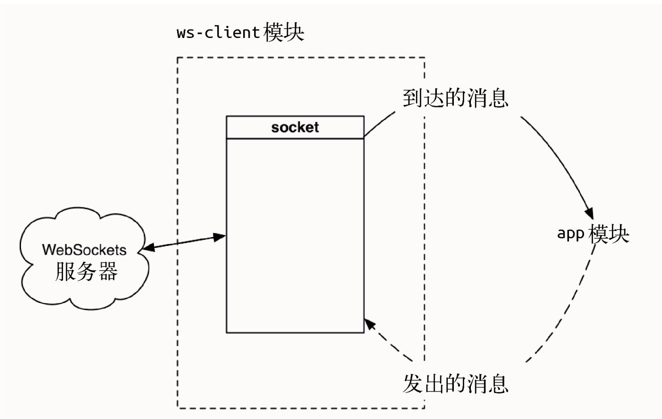


图17-14　ws-client接口

在构建客户端的过程中，我们还将认识一些ES6的新特性。

17.6.1　处理连接

首先创建连接的处理程序。打开ws-client.js文件，给WebSocket连接声明一个变量。

```
let socket;
```

上面的声明使用了ES6中定义变量的新方法，叫作**let作用域**。假如你使用let作用域来声明一个变量——关键字不是var，而是let——那么变量将不会被**提升**（hoist）。

提升的意思是，变量声明被移动到创建这些变量的函数作用域的开头，这属于JavaScript解析器在后台执行的操作。不幸的是，这些操作会导致一些难以察觉的错误。

在本章最后会介绍更多有关提升的知识，现在只需要知道：在if/else语句和循环体里，let是一种更安全的声明变量的方式。

现在，向ws-client.js文件里添加一个方法，用来初始化连接。

```
let socket;

function init(url)  {
  socket = new WebSocket(url);
  console.log('connecting...');
}
```

init函数连接到WebSocket服务器。接下来，要把ws-client.js导入到app.js的ChatApp中。

为了能被调用，ws-client.js要指定导出的内容。这里需要导出一个单独的值：一个将导出的函数作为属性值的对象。跟本章开头一样，使用export default语法——加上额外的ES6简写方法。

将export添加到ws-client.js的最后，如下所示：

```
...
function init(url) {
  socket = new WebSocket(url);
  console.log('connecting...');
}

export default {
  init,
}
```

注意，不需要指定属性名称。这种语法缩写等价于：

```
export default {
  init: init
}
```

假如键和值名称一样，ES6允许省略冒号和值。键会自动作为变量名，值会自动关联到变量名对应的值。这个ES6特性是**增强版的对象字面量**语法。

现在ws-client模块已经创建完毕，接下来要在app.js中导入ws-client模块提供的值。首先在app.js的开头添加一个导入声明：

```
import socket from './ws-client';

class ChatApp {
  constructor() {
    console.log('Hello ES6!');
  }
}
  ...
```

socket就是从ws-client.js中导出的对象。

接下来，在ChatApp构造函数中调用socket.init方法，将WebSocket服务器的URL传进去。

```
import socket from './ws-client';

  class ChatApp {
    constructor() {
      console.log('Hello ES6!');
      socket.init('ws://localhost:3001');
    }
  }
  ...
```


npm脚本会重新编译代码。（假如已经停止了npm run watch或者npm run dev，需要在不同的窗口重启这两个脚本。）重新加载浏览器，会看到控制台打印出'connecting...'，如图17-15所示。

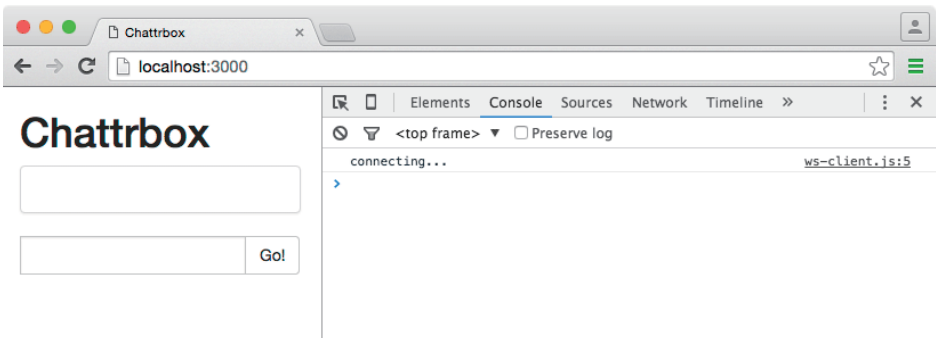


图17-15　WebSocket初始化时打印出来的消息。

现在已经启动并开始运行应用程序主体部分了。

17.6.2　处理事件并发送消息

当App模块调用init方法时，会实例化一个新的WebSocket对象，并与服务器建立一个连接。但是App模块需要感知到该操作已经完成，以便在连接上进行一些操作。

WebSocket对象提供了一系列专门用于处理事件的属性，其中一个是onopen属性。当与WebSocket服务器的连接成功建立时，就会调用赋给这个属性的函数。在这个函数中，可以对连

接进行任何操作。

为了让ws-client模块灵活且可复用，请不要将App模块需要对连接执行的操作进行硬编码，而是使用与在CoffeeRun中注册点击和提交处理程序时一样的模式。

给ws-client.js添加一个registerOpenHandler函数。registerOpenHandler接受一个回调函数作为参数，给onopen赋一个函数，然后在这个onopen函数中调用回调函数。

```
let socket;

function init(url) {
  socket = new WebSocket(url);
  console.log('connecting...');
}

function registerOpenHandler(handlerFunction) {
  socket.onopen = () => {
    console.log('open');
    handlerFunction();
  };
}
...
```

onopen函数定义跟以前写的不太一样。这种方式是ES6的一种新语法，叫做**箭头函数**。箭头函数是匿名函数的一种缩写方式。除了写起来更简单，箭头函数与一般的匿名函数一模一样。

registerOpenHandler接受一个函数参数（handlerFunction），并给socket连接的onopen属性赋值为一个匿名函数。在这个匿名函数内，调用参数handlerFunction。

（使用匿名函数比写socket.onopen = handlerFunction要复杂一些。当需要响应一个事件但需要执行一些中间操作时——比如打印日志消息——使用匿名函数效果更好。）

接下来需要写一个接口，用来处理经由WebSocket连接收到的消息。在ws-client.js上新增一个registerMessageHandler方法。将socket的onmessage属性赋值为一个箭头函数，该箭头函数接受一个事件参数。

```
...
function registerOpenHandler(handlerFunction) {
  socket.onopen = () => {
    console.log('open');
    handlerFunction();
  };
}

function registerMessageHandler(handlerFunction) {
  socket.onmessage = (e) => {
    console.log('message', e.data);
    let data = JSON.parse(e.data);
    handlerFunction(data);
  };
}
...
```

箭头函数的参数位于括号内，和常规函数一样。

在registerMessageHandler内，Chattrbox客户端程序能通过onmessage回调函数接受服务端发来的一个对象。该对象表示消息事件，它有一个data属性，包含了服务端发送的JSON字符串。每收到一个字符串，就需要将字符串转换成JavaScript对象，然后将其发送给handler-Function。

还需要最后一部分代码，用来将消息发送给WebSocket。在ws-client.js中增加一个sendMessage函数。发送消息总共分两步：首先将返回的消息内容（包括消息内容、用户名和时间戳）转换成JSON字符串，然后将JSON字符串发送给WebSocket服务器。

```
...
function registerMessageHandler(handlerFunction) {
  socket.onmessage = (e) => {
    console.log('message', e.data);
    let data = JSON.parse(e.data);
    handlerFunction(data);
  };
}

function sendMessage(payload) {
  socket.send(JSON.stringify(payload));
}
...
```


最后，使用增强版的对象字面量语法将新增方法导出。

```
  ...
  function sendMessage(payload) {
    socket.send(JSON.stringify(payload));
  }

  export default {
    init,
    registerOpenHandler,
    registerMessageHandler,
    sendMessage
}
```

现在ws-client.js已经具备了所有跟服务端通信的方法。最后一项工作就是测试ws-client.js：用这个模块发送一条消息。

17.6.3　发出和回应一条消息

在app.js里更新ChatApp构造函数。调用socket.init之后，调用registerOpenHandler和registerMessageHandler，为它们传递箭头函数。

```
import socket from './ws-client';

class ChatApp {
  constructor() {
    socket.init('ws://localhost:3001');
```

```
      socket.registerOpenHandler(() => {
        let message = new ChatMessage({ message: 'pow!' });
        socket.sendMessage(message.serialize());
      });
      socket.registerMessageHandler((data) => {
        console.log(data);
      });
    }
  }
  ...
```

建立连接之后，立即发送一条模拟消息。收到消息后，控制台打印出消息。

保存代码，在编译完成后刷新浏览器，就能看到发送和回应的消息了（如图17-16所示）。

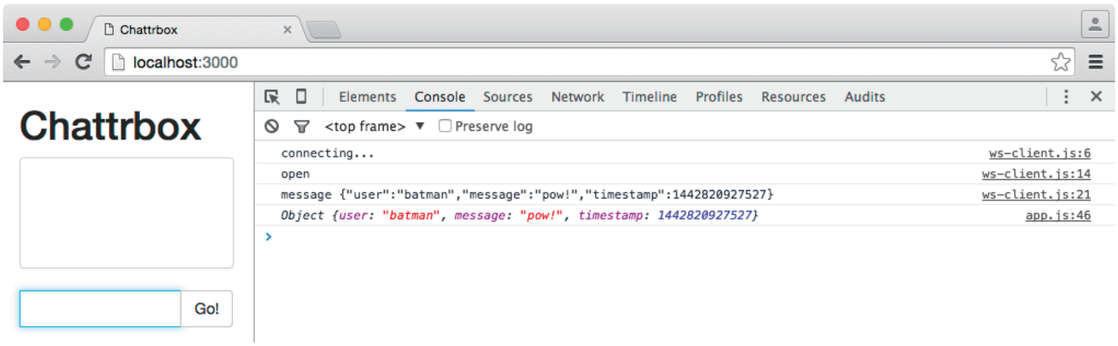


图17-16　通过WebSocket调用和响应

简直太棒了！现在已经完成了Chattrbox的三个主要模块中的两个了，下一章将会完成Chattrbox的全部功能。在下一章中，我们将会创建一个模块，这个模块负责将已有的模块关联到UI中。该模块会将新消息渲染到消息列表，并在提交表单时将消息发出。

17.7　延展阅读：将其他语言编译成 JavaScript

有少数语言能够被编译成JavaScript。下面有一个简短的列表：

- CoffeeScript：coffeescript.org
- TypeScript：www.typescriptlang.org
- C/C++：kripken.github.io/emscripten-site

其中最重要的是CoffeeScript，它提供了一些最常见模式的简写语法（比如匿名函数的箭头语法）。实际上，CoffeeScript对ES6有举足轻重的影响。

Google、MicroSoft、Mozilla还有其他公司合作创建了一个项目，用来标准化一个用于JavaScript引擎的assembly语言。该项目叫作WebAssembly，目标是创建一种由多种语言编译而成的高性能低级语言。

WebAssembly的目的是补充JavaScript（而不是取代它）并利用多种语言的优点。比如，

JavaScript擅长于创建基于浏览器的应用程序，但是并不擅长渲染数学密集型的游戏图像；而C和C++则极其擅长渲染游戏代码。与其将C++代码移植成JavaScript并且造成潜在的bug，倒不如将其编译成WebAssembly。

WebAssembly项目发源于一个早期项目asm.js。asm.js项目定义了JavaScript的一个子集，旨在编写高性能的代码。

更多有关asm.js和WebAssembly的信息请查看JavaScript之父的这篇文章：brendaneich.com/2015/06/from-asm-js-to-webassembly。

17.8 初级挑战：默认导入名称

你在main.js中导入声明，创建了一个叫作`ChatApp`的本地变量。假如把变量名改成`ApplicationForChatting`会怎样呢？

试一下（但是要确保下一行的`new`声明也改成新的变量名），看看是不是还能生效。如果能，为什么？如果不能，为什么不能？

17.9 中级挑战：提醒连接关闭

在`ws-client`模块中添加一个新的函数`registerCloseHandler`。这个函数接受一个回调函数，当在socket上触发关闭事件时，调用这个回调函数。

在main.js中用`registerCloseHandler`提醒用户连接被关闭了，然后测试函数是否生效。

怎样测试呢？显然不能关闭浏览器窗口，因此需要关闭连接的另一端。

再交给你一个额外任务：写一个函数来尝试重连。可以用一个`setTimeout`或者请求用户的确认（在MDN上搜索更多细节）。

17.10 延展阅读：变量提升

JavaScript的问世使得非专业程序员也可以创建一些具有基本交互性的Web内容。尽管这门语言有些特性旨在让代码具备抗错误性，但是有些特性在实践中却容易造成错误，其中之一便是提升。

当JavaScript引擎解析代码的时候，它会查找所有的变量和函数声明，将它们移到其所在函数的顶部。（假如它们不在函数内，则会在其余代码之前被计算。）

示例能给出最好的说明。如下代码：

```
function logSomeValues () {
  console.log(myVal);
  var myVal = 5;
  console.log(myVal);
}
```

将会被解析成下面这样：

```
function logSomeValues () {
  var myVal;
  console.log(myVal);
  myVal = 5;
  console.log(myVal);
}
```

如果在控制台调用logSomeValues，将会看到下面这样的输出：

```
> logSomeValues();
Undefined
5
```

注意，只有**声明**被提升了，赋值操作还在原地。这自然会让人迷惑，尤其当你试图在if或者循环里声明变量时。在其他语言中，大括号表示一个**代码块**，拥有自己的作用域。在JavaScript中，大括号并不会创建作用域，只有函数才创建作用域。

来看另一个示例：

```
var myVal = 11;
function doNotWriteCodeLikeThis() {
  if (myVal > 10) {
    var myVal = 0;
    console.log('myVal was greater than 10; resetting to 0');
  } else {
    console.log('no need to reset.');
  }
  return myVal;
}
```

你的预期也许是在控制台打印出'myVal was greater than 10; resetting to 0'，并且返回值为0。但是，真实的输出如下：

```
> doNotWriteCodeLikeThis();
no need to reset.
undefined
```

var myVazl声明被移到函数顶部，所以在执行if语句之前，myVal的值为undefined。赋值操作仍然保持在if代码块中。

函数声明也会被提升，不过它是被整体提升。也就是说下面的代码会正常运行：

```
boo();

// 在调用之后声明:
function boo() {
  console.log('BOO!!');
}
```

JavaScript将整个函数声明块移动到顶部，调用boo不会有任何问题：

```
> boo();
BOO!!
```

提升并不会移动let声明，同样也不会移动const声明，它用于声明无法重新赋值的变量。

17.11 延展阅读：箭头函数

我收回前面的话：箭头函数的功能并不**完全**等同于匿名函数。在某些情况下，箭头函数**更好**。除了提供简短的语法，箭头函数有以下优点。

- 可以实现`function(){}.bind(this)`的效果，让`this`指向箭头函数本身所处的作用域。
- 假如只有一句声明的话，可以省略大括号。
- 在省略大括号的情况下，会返回这一句声明的结果。

举个例子，这里是CoffeeRun的`CheckList.prototype.addClickHandler`方法：

```
CheckList.prototype.addClickHandler = function(fn) {
  this.$element.on('click', 'input', function (event) {
    var email = event.target.value;
    fn(email)
      .then( function () {
        this.removeRow(email);
      }.bind(this));
  }.bind(this));
};
```

将这里的匿名函数替换成箭头函数，会使代码简洁一些：

```
CheckList.prototype.addClickHandler = (fn) => {
  this.$element.on('click', 'input', (event) => {
    let email = event.target.value;
    fn(email)
      .then(() => this.removeRow(email));
  });
};
```

去掉多余的`function`和`.bind(this)`后，`addClickHandler`的功能更突出了。

第 18 章 继续ES6探索之旅

Chattrbox是真正能使用的应用，但是目前只实现了“底层”业务逻辑：连接WebSocket服务器、定义消息格式以及发送和接收消息。

本章将完成Chattrbox的最后一部分：添加UI层。这个过程将继续使用Node和npm管理编译程序，提供服务端应用。等到本章结束，一个全功能的Web聊天应用将会重磅登场（如图18-1所示）。

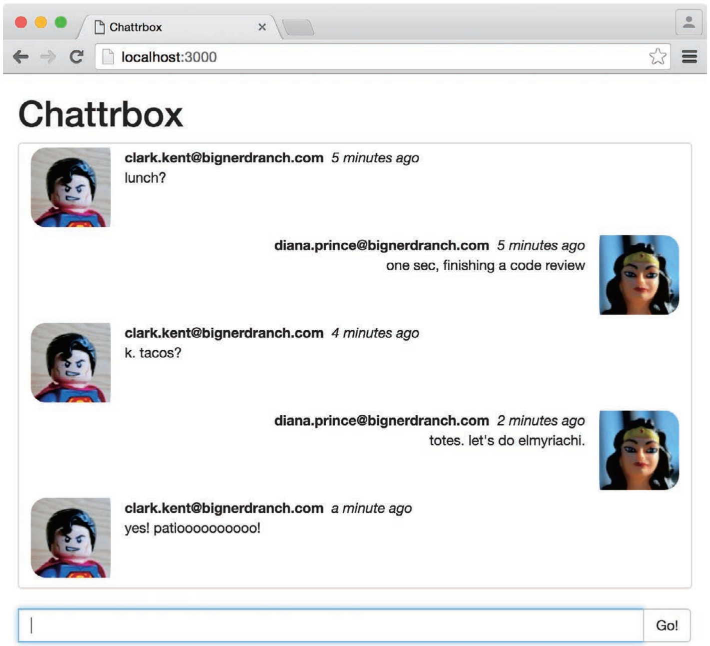


图18-1　完成后的Chattrbox

在构建CoffeeRun时，创建了`FormHandler`和`CheckList`模块，这两个模块分别对应表单和列表区域。Chattrbox将使用相同的模式，创建`ChatForm`和`ChatList`模块。

还要创建一个`UserStore`模块，用来保存当前聊天用户的信息。这些模块能让Chattrbox程序更健壮，并且让主要模块拥有更好的复用性。

18.1　将 jQuery 安装成一个 Node 模块

Chattrbox将使用jQuery进行DOM操作。但是这次不会像CoffeeRun那样从cdnjs.com去加载jQuery，也不会像以前集成客户端依赖那样在HTML中使用`<script>`标签。

有了Browserify，这些都不需要了——Browserify会自动将JavaScript依赖项构建到浏览器使用的应用程序包中。所以，现在如果想集成jQuery，只需要通过`import`包含它，Browserify就会处理剩下的事情。

首先，安装jQuery库到node_modules文件夹：

```
npm install --save-dev jquery
```

打开dom.js文件，开始编写该模块代码。dom.js模块将会用到jQuery，所以添加一句`import`声明以包含它。

```
import $ from 'jquery';
```

稍后将会安装并使用另一个第三方库，安装和导入的步骤同上。

18.2　创建 ChatForm 类

就像在CoffeeRun中一样，创建一个对象来管理DOM里的表单元素。这里通过`ChatForm`类来实现。使用ES6的类会让代码比CoffeeRun里的代码可读性更强一些。

创建一个`ChatForm`实例和初始化事件处理程序是两个单独的步骤，因为构造函数的职责应该仅仅是设置实例的属性，其他任务（比如添加事件监听器）应该在别的方法中实现。

在dom.js中定义`ChatForm`，它的构造函数接受选择器参数。在这个构造函数中，为实例需要追踪的元素添加属性。

```
import $ from 'jquery';

class ChatForm {
  constructor(formSel, inputSel) {
    this.$form = $(formSel);
    this.$input = $(inputSel);
  }
}
```

然后，添加一个`init`方法，为表单的`submit`事件关联一个回调函数。

```
...
class ChatForm {
  constructor(formSel, inputSel) {
    this.$form = $(formSel);
    this.$input = $(inputSel);
  }

  init(submitCallback) {
    this.$form.submit((event) => {
      event.preventDefault();
```

```
      let val = this.$input.val();
      submitCallback(val);
      this.$input.val('');
    });

    this.$form.find('button').on('click', () => this.$form.submit());
  }
}
```

init方法中的提交事件处理程序用到了箭头函数。该箭头函数阻止了表单默认的提交行为，取得表单输入框的值并将其传递给submitCallback，最终重置输入框的值。

为确保点击按钮时提交表单，添加一个点击事件处理程序，让表单触发submit事件。为实现这一过程，用jQuery获取表单元素，然后调用jQuery的submit方法。这里用到了箭头函数的单个表达式版本，可以省掉大括号。

为使用该模块，需要export ChatForm。前一章使用了export default来实现模块导出，这种方式允许导出模块的单个值。在某些情况下，还可以在单个默认值里使用一个简单的JavaScript对象来封装多个值。

本章将使用**命名的导出**（named export）方式来导出多个命名值，而不是导出一个默认值。

通过在class声明前面添加export关键字，导出ChatForm类。当用户使用该模块时，便可通过类名访问到它。

```
...
export class ChatForm {
  constructor(formSel, inputSel) {
    this.$form = $(formSel);
    this.$input = $(inputSel);
  }
...
```

很简单吧！现在将ChatForm导入到app.js中。

你在Ottergram和CoffeeRun中使用了var关键字来表示选择器字符串。而在ES6中，可以通过声明常量来实现这一目的，因为字符串的值不会改变。就像let一样，const也是块级作用域的，也就是说它可以被同个大括号里的任何代码访问到。当常量位于所有大括号外面时（就像这里的示例一样），它就可以被同个文件里的任何代码访问到。

在app.js中导入ChatForm类，为表单选择器和消息输入框选择器创建常量。同时，在ChatApp的构造函数中创建一个ChatForm的实例。

```
import socket from './ws-client';
import {ChatForm} from './dom';

const FORM_SELECTOR = '[data-chat="chat-form"]';
const INPUT_SELECTOR = '[data-chat="message-input"]';

class ChatApp {
  constructor() {
    this.chatForm = new ChatForm(FORM_SELECTOR, INPUT_SELECTOR);
```

```
      socket.init('ws://localhost:3001');
      socket.registerOpenHandler(() => {
        let message = new ChatMessage('pow!');
        socket.sendMessage(message.serialize());
      });
      socket.registerMessageHandler((data) => {
        console.log(data);
      });
    }
  }
...
```

导入ChatForm时，用大括号将它包起来——{ChatForm}——就成为了一个**命名的导入**。这里ChatForm的命名的导入声明了一个本地变量ChatForm，并将其绑定到dom模块中同名变量的值上。

将ChatForm连接到socket

在上一章中，你发送了一条模拟消息"pow!"。现在可以通过ChatForm发送真正的表单数据了。

在socket.registerOpenHandler回调函数中初始化ChatForm的实例。一定要在socket连接打开**之后**初始化，而不能一创建实例就初始化。这种等待能避免用户输入了聊天消息却不能发送到服务器的情况。（毕竟如果消息发不出去，用户体验会很糟糕。）

记得要给ChatForm的init方法传一个回调函数，用来处理表单的提交。

在app.js中删除模拟数据，改成调用ChatForm.init。给它传一个回调函数，将来自ChatForm的消息数据发给socket。

```
...
class ChatApp {
  constructor() {
    this.chatForm = new ChatForm(FORM_SELECTOR, INPUT_SELECTOR);

    socket.init('ws://localhost:3001');
    socket.registerOpenHandler(() => {
      let message = new ChatMessage('pow!');
      socket.sendMessage(message.serialize());
      this.chatForm.init((data) => {
        let message = new ChatMessage({message: data});
        socket.sendMessage(message.serialize());
      });
    });
    socket.registerMessageHandler((data) => {
      console.log(data);
    });
  }
}
...
```

再来看看ChatApp都做了些什么。首先，它打开与服务端的socket连接。连接打开后，ChatApp

初始化ChatForm的实例，并传了一个表单提交的回调函数。

现在,当用户在表单提交一条消息时,ChatForm实例会获取消息数据,并将其发送给ChatApp里的回调函数。回调函数会将消息打包成一个ChatMessage，并发送给WebSocket服务器。

18.3　创建 ChatList 类

前面的ChatForm负责向外发送聊天消息。下一项工作是在服务端发来新消息时，将其展示出来。因此要在dom.js中创建第二个类，用来向用户展示聊天消息列表。

ChatList会给每条消息创建DOM元素，以展示发送该消息的用户名以及消息内容。在dom.js中，创建并导出类ChatList的定义来实现该功能：

```
import $ from 'jquery';

export class ChatForm {
  ...
}

export class ChatList {
  constructor(listSel, username) {
    this.$list = $(listSel);
    this.username = username;
  }
}
```

ChatList接受属性选择器和用户名作为参数。属性选择器用于决定将创建的消息列表元素添加到哪个元素，用户名用于区分发送消息的是当前用户还是其他人。（当前用户的消息和别人发送的消息会区分展示。）

现在ChatList已经具备了一个构造函数，它还需要为消息创建DOM元素。

给ChatList添加一个drawMessage方法。它接受一个对象参数，并将该参数解构成本地变量，用来表示用户名、时间戳以及消息内容。（为了解释解构赋值，下面的例子使用单字符的本地变量。）

```
...
export class ChatList {
  constructor(listSel, username) {
    this.$list = $(listSel);
    this.username = username;
  }

drawMessage({user: u, timestamp: t, message: m}) {
  let $messageRow = $('<li>', {
    'class': 'message-row'
  });

  if (this.username === u) {
    $messageRow.addClass('me');
  }
```

```
    let $message = $('<p>');

    $message.append($('<span>', {
      'class': 'message-username',
      text: u
    }));

    $message.append($('<span>', {
      'class': 'timestamp',
      'data-time': t,
      text: (new Date(t)).getTime()
    }));

    $message.append($('<span>', {
      'class': 'message-message',
      text: m
    }));

    $messageRow.append($message);
    this.$list.append($messageRow);
    $messageRow.get(0).scrollIntoView();
  }
}
```


drawMessage方法创建一行消息，包括用户名、时间戳和消息内容本身。假如当前用户是消息的发送者，对应的消息元素会有一个额外的CSS类名用来区分样式。然后将消息行添加到ChatList的列表元素中，并且将新消息行滚动到可视区域。

至此，ChatList已经完成。现在将它整合到ChatApp中。

在app.js中更新dom的导入声明，用以导入ChatList。添加一个const用来表示列表选择器，然后在构造函数中实例化一个新的ChatList。

```
import socket from './ws-client';
import {ChatForm, ChatList} from './dom';

const FORM_SELECTOR = '[data-chat="chat-form"]';
const INPUT_SELECTOR = '[data-chat="message-input"]';
const LIST_SELECTOR = '[data-chat="message-list"]';

class ChatApp {
  constructor() {
    this.chatForm = new ChatForm(FORM_SELECTOR, INPUT_SELECTOR);
    this.chatList = new ChatList(LIST_SELECTOR, 'wonderwoman');

    socket.init('ws://localhost:3001');
    ...
```

马上就能完成基本的聊天功能了。最后一步是在新消息到达时调用chatList.drawMessage进行绘制，这一步在app.js里的registerMessageHandler中实现。

```
...
class ChatApp {
  ...
    socket.registerMessageHandler((data) => {
      console.log(data);
      let message = new ChatMessage(data);
      this.chatList.drawMessage(message.serialize());
    });
  }
}
...
```

用接收的数据创建一个新的`ChatMessage`，然后对消息进行序列化。这一步属于预防措施，用于去除数据里可能存在的多余元数据。根据socket数据创建一个新的`ChatMessage`用以提供消息，然后用`this.chatList.drawMessage`将序列化之后的消息绘制到浏览器中。

现在运行一下代码。假如还没有编译，启动Watchify（用`npm run watch`）和`nodemon`（用`npm run dev`）。打开或者刷新浏览器，输入消息（如图18-2所示）。

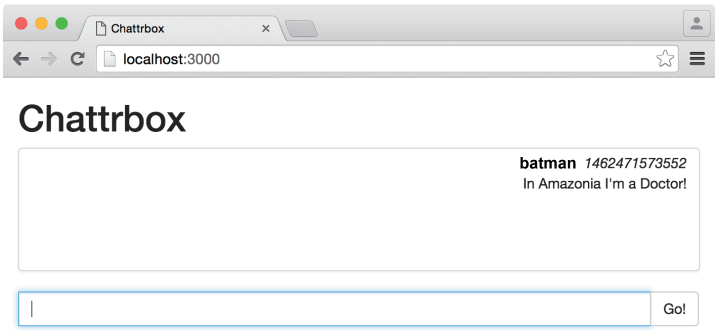


图18-2　看，你自己的聊天消息

太棒了！现在终于有一个真正可用的聊天应用了。不过，还需要加一点设计元素润色一番。

18.4　使用 Gravatar

Gravatar是一个可用于关联头像和邮箱地址的免费服务，它通过一个专门格式化的URL提供每个用户的头像。比如，图18-3就是一个测试账号的头像。

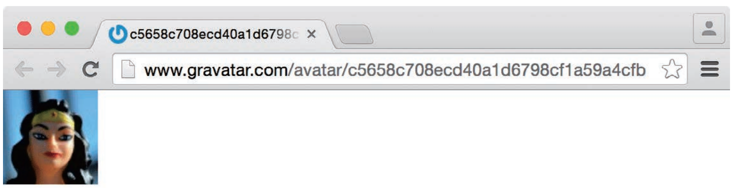


图18-3　Gravatar图片示例

看到URL的最后一部分了吗？这部分是根据用户邮箱地址生成的唯一标识。这个标识叫作**哈希**（hash），用第三方库`crypto-js`很容易生成。

用npm将`crypto-js`添加到项目中：

```
Add crypto-js to your project using npm:

npm install --save-dev crypto-js
```

`crypto-js`现在已经安装在项目本地的node_modules文件夹下，可供随时使用。

用JavaScript创建字符串时，经常需要将字符串与其他值进行拼接。为了创建包含表达式和变量值的字符串，ES6提供了更好的方法——**模板字符串**。接下来使用这个功能创建访问Gravatar图片的URL。

在dom.js中添加另一个`import`声明，导入`crypto-js`库的子模块`md5`，使用`/`来分隔主模块和子模块的名称。然后，写一个`createGravatarUrl`函数，它接受一个用户名，用来生成一个MD5的哈希值，并返回Gravatar的URL。


```
import $ from 'jquery';
import md5 from 'crypto-js/md5';

function createGravatarUrl(username) {
  let userhash = md5(username);
  return `http://www.gravatar.com/avatar/${userhash.toString()}`;
}
...
```

注意：在`` return `http://www.gravatar.com/avatar/${userhash.toString()}` ``中的符号并不是单引号，而是反引号，大多数美式键盘的Escape键的下面就是这个键。

在反引号里使用`${userhash.toString()}`语法，就可以直接在字符串中包含JavaScript表达式的值。在这个例子的表达式里引用了变量`userhash`并调用了它的`toString`方法，实际上，大括号中可以包含任何表达式。

接下来，使用这个函数在新消息中展示Gravatar。在`ChatList`的底部，`drawMessage`方法（还是在dom.js中）创建了一个新的图片元素，将`src`属性设置成用户的Gravatar。

```
...
    $message.append($('<span>', {
      class: 'message-message',
      text: m
    }));

    let $img = $('<img>', {
      src: createGravatarUrl(u),
      title: u
    });

    $messageRow.append($img);
    $messageRow.append($message);
    this.$list.append($messageRow);
    $messageRow.get(0).scrollIntoView();
...
```

运行聊天应用，这次看到出现了一个Gravatar头像（如图18-4所示）。

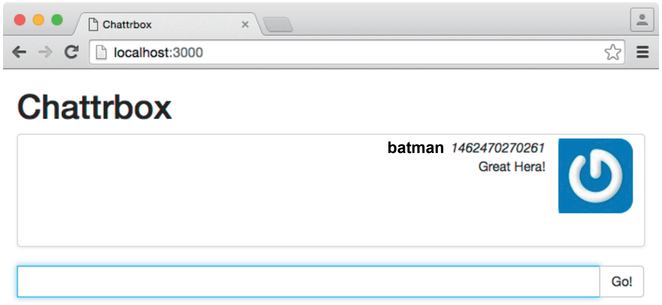


图18-4　显示一个Gravatar头像

很遗憾，wonderwoman这个用户名并没有对应的Gravatar头像，所以只能看到一个长相平平的默认Gravatar头像。

18.5　请求用户名

尽管成为神奇女侠（Wonder Woman）非常酷，但是成为一个使用Chattrbox的JavaScript开发者更酷。（而且真实用户真的拥有Gravatar头像。）为了知道谁在用Chattrbox，需要请求用户输入他们的用户名。

这一功能需要dom模块与UI进行交互，因此在dom.js中创建一个`promptForUsername`函数，将其添加到`export`，而不是作为`ChatForm`或者`ChatList`的一部分。

```
...
function createGravatarUrl(username) {
  let userhash = md5(username);
  return `http://www.gravatar.com/avatar/${userhash.toString()}`;
}

export function promptForUsername() {
  let username = prompt('Enter a username');
  return username.toLowerCase();
}
...
```

在`promptForUsername`函数中创建一个`let`变量，用以保存用户输入的文字。（`prompt`函数是浏览器内置的，它会返回一个字符串。）然后返回转换成小写格式的文字。

接下来更新app.js，在其中使用刚添加的函数。更新`dom`模块的`import`声明，调用`promptForUsername`函数，获取`username`变量的值：

```
import socket from './ws-client';
import {ChatForm, ChatList, promptForUsername} from './dom';

const FORM_SELECTOR = '[data-chat="chat-form"]';
```

```
const INPUT_SELECTOR = '[data-chat="message-input"]';
const LIST_SELECTOR = '[data-chat="message-list"]';

let username = '';
username = promptForUsername();

class ChatApp {
  ...
```

现在更新ChatMessage，使用刚才拿到的用户名作为默认用户名。记住，只有从服务端取得的消息才有data.user值。

```
...
class ChatMessage {
  constructor({
    message: m,
    user: u='batman', username,
    timestamp: t=(new Date()).getTime()
...
```

最后将用户名传给ChatList构造函数：

```
...
class ChatApp {
  constructor() {
    this.chatForm = new ChatForm(FORM_SELECTOR, INPUT_SELECTOR);
    this.chatList = new ChatList(LIST_SELECTOR, 'wonderwoman' username);
    ...
```

构建完毕后，刷新浏览器，在请求输入框中输入用户名（如图18-5所示）。

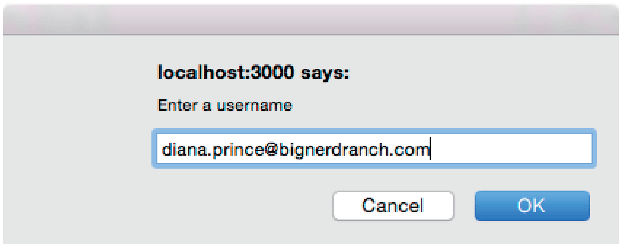


图18-5　请求用户名

现在尝试发送消息，可以看到之前挑选的用户名会被服务端返回，同时返回的还有与之关联的Gravatar头像（如图18-6所示）。

Gravatar头像是通过邮箱地址关联的。假如你的邮箱地址没有关联头像，试试diana.prince@bignerdranch.com或者clark.kent@bignerdranch.com。

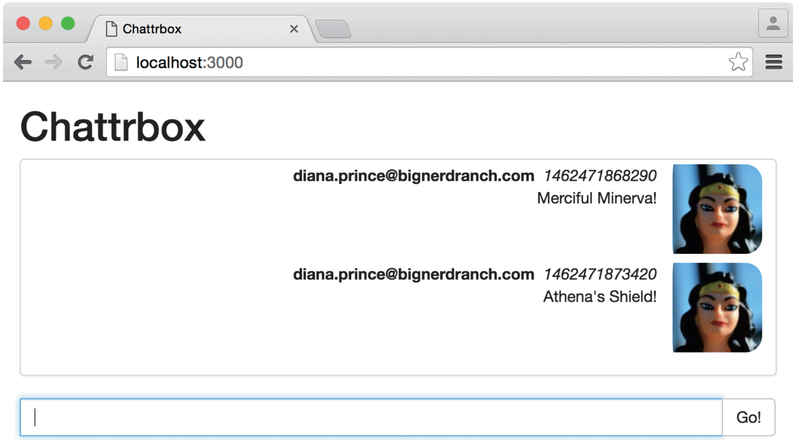


图18-6　用户名

18.6　使用会话存储

每次刷新页面都要输入用户名实在是太麻烦了，要是能将用户名存储在浏览器就会好很多。为了实现简单的存储，浏览器提供了两个API用来存储键值对（有一个限制——值必须是字符串）。这两个API是localStorage和sessionStorage。在localStorage和sessionStorage中存储的数据与Web应用服务器地址相关联。不同网站的代码无法访问彼此的数据。

使用localStorage没问题，但是你可能只想将用户名保存到关闭浏览器的标签或者窗口为止。在这种情况下，就要用sessionStorage API了。它跟localStorage很像，但是在浏览器会话结束时数据会被清除（不管是关闭浏览器的标签还是窗口）。

接下来创建一系列新的类来管理sessionStorage信息。

在app/scripts/src文件夹下新建一个storage.js文件，并定义一个新的类：

```
class Store {
  constructor(storageApi) {
    this.api = storageApi;
  }
  get() {
    return this.api.getItem(this.key);
  }

  set(value) {
    this.api.setItem(this.key, value);
  }
}
```

新的Store类是通用类，它既可以搭配localStorage使用，也可以搭配sessionStorage使用。它只是简单地封装了一下Web Storage API。在实例化这个类的时候，可以指定使用哪个Storage API。

注意，**没有**在构造函数中设置对this.key的引用，因为Store类并不需要给自己提供存储的数据。相反，它用来创建定义key属性的子类。

使用extends关键字创建一个子类，用于在sessionStorage中保存用户名：

```
class Store {
  constructor(storageApi) {
    this.api = storageApi;
  }
  get() {
    return this.api.getItem(this.key);
  }

  set(value) {
    this.api.setItem(this.key, value);
  }
}

export class UserStore extends Store {
  constructor(key) {
    super(sessionStorage);
    this.key = key;
  }
}
```


UserStore只定义了一个构造函数，它执行两个操作。首先，调用super，这一步会调用Store的构造函数，并传入一个对sessionStorage的引用。然后，给this.key设置一个值。

现在Store类的api的值已经设置好了，UserStore实例的key的值也设置好了。所有代码已经准备就绪，UserStore实例可以调用get和set方法了。

app.js将会用到UserStore，所以需要把它导出。

现在来使用新的UserStore。将UserStore导入到app.js中，创建一个实例，用这个实例来存储用户名：

```
import socket from './ws-client';
import {UserStore} from './storage';
import {ChatForm, ChatList, promptForUsername} from './dom';

const FORM_SELECTOR = '[data-chat="chat-form"]';
const INPUT_SELECTOR = '[data-chat="message-input"]';
const LIST_SELECTOR = '[data-chat="message-list"]';

let username = '';
let userStore = new UserStore('x-chattrbox/u');
let username = userStore.get();
if (!username) {
  username = promptForUsername();
  userStore.set(username);
}

class ChatApp {
  ...
```

再次在浏览器中运行Chattrbox。这一次，只需要在初次加载页面的时候提交用户名了，之后刷新都会用第一次输入的用户名。

为了确保用户名已经存储到`sessionStorage`中，使用开发者工具里的Resources面板。点击Resources面板后，会看到左边有一个列表。在列表中点击Session Storage左边的三角形▶，就会展开显示http://localhost:3000。点击这个URL，就能看到`UserStore`存储的数据（如图18-7所示）。

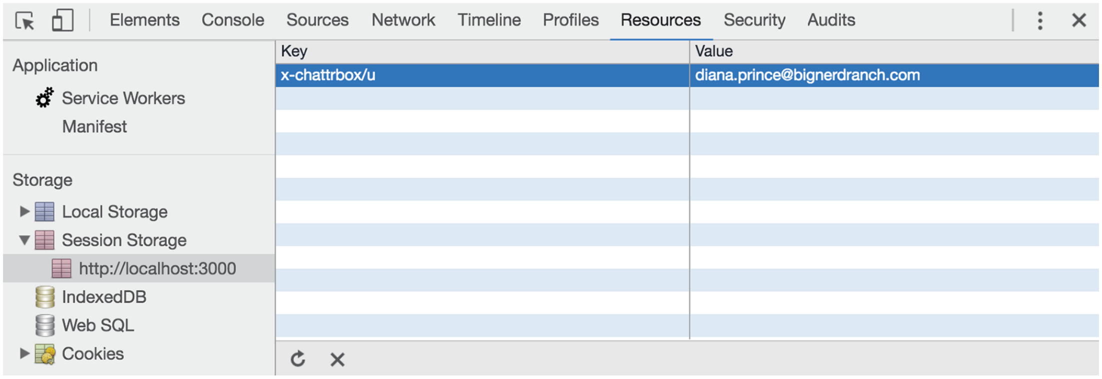


图18-7　开发者工具里的Resources面板

在右侧键值对列表的底部有两个按钮，可以分别用来刷新列表和删除列表中的元素。假如你想手动修改存储的数据，可以使用这两个按钮。

18.7　格式化和更新消息时间戳

现在的消息时间戳对用户来说并不友好。（讲真的，谁会用从1970年1月1日开始的毫秒数来表示时间？）为了提供更友好的时间戳（比如“10分钟之前”），需要增加一个叫作`moment`的模块。用npm安装这个模块，将其保存为开发环境下的依赖包。

```
npm install --save-dev moment
```

每条消息都将时间戳存储成了数据属性。为`ChatList`写一个`init`方法，用来调用内置函数`setInterval`。这个函数接受两个参数：要运行的函数，和这个函数多久运行一次。这个函数将会更新每条消息，将时间戳转换成用户可读的格式。

为了设置时间戳字符串，在dom.js里用jQuery查找所有带有`data-time`属性的元素，这个属性的值都是数字化的时间戳。使用这个数字化的时间戳创建一个新的`Date`对象，将该对象传给`moment`。然后调用`fromNow`方法处理最终的时间戳字符串，并将结果字符串设为元素的HTML文本。

```
import moment from 'moment';
...
  drawMessage({user: u, timestamp: t, message: m}) {
    ...
  }

  init() {
    this.timer = setInterval(() => {
      $('[data-time]').each((idx, element) => {
        let $element = $(element);
        let timestamp = new Date().setTime($element.attr('data-time'));
        let ago = moment(timestamp).fromNow();
        $element.html(ago);
      });
    }, 1000);
  }
}
```


这个函数会每1000毫秒执行一次。为了保证用户可以马上看到一个可读的时间戳，更新`drawMessage`。在初次将消息绘制到聊天列表时，使用`moment`创建一个格式化的时间戳字符串。

```
...
  drawMessage({user: u, timestamp: t, message: m}) {
    ...
    $message.append($('<span>', {
      'class': 'timestamp',
      'data-time': t,
      text: (new Date(t)).getTime()
      text: moment(t).fromNow()
    }));
    ...
```

最后更新app.js，在`socket.registerOpenHandler`的回调函数中调用`this.chatList. init`：

```
...
class ChatApp {
  constructor () {
    this.chatForm = new ChatForm(FORM_SELECTOR, INPUT_SELECTOR);
    this.chatList = new ChatList(LIST_SELECTOR, username);

    socket.init('ws://localhost:3001');
    socket.registerOpenHandler(() => {
      this.chatForm.init((text) => {
        let message = new ChatMessage({message: text});
        socket.sendMessage(message.serialize());
      });
      this.chatList.init();
    });
  ...
```

保存代码，让npm脚本去编译所有改动。刷新浏览器，开始聊天。你会看到消息文本使用了新的时间戳格式。几分钟后，还能看到消息时间戳更新了（如图18-8所示）。

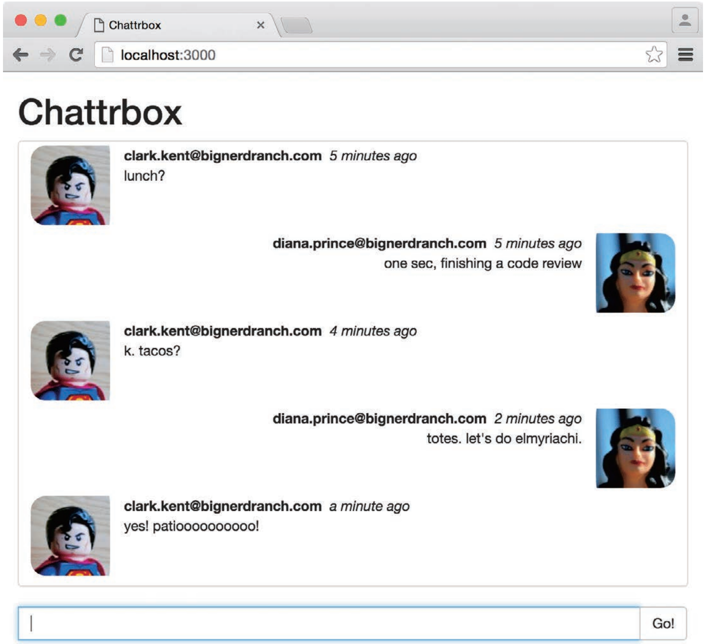


图18-8　不是很私密的标识

Chattrbox项目已经开发完毕。尽管只用了几章，但是确实收获了一些令人兴奋的成果。你学会了用Node.js写两种服务器：一个基础的Web服务器和一个WebSocket服务器。你还用ES6构建了客户端应用程序，利用Babel和Browserify将代码编译成ES5，从而在较旧的浏览器中使用Chattrbox。而且你还用npm脚本实现了工作流的自动化。

Chattrbox是你目前所学的技术巅峰。在下一个项目Tracker中，将会介绍Ember.js——一个用来构建大型应用的框架。它将基于你所学的模块化、异步编程和工作流工具的知识进行构建。

18.8　初级挑战：给消息添加特效

给新消息添加一个特效，比如让消息渐入或者划入。（阅读jQuery的Effects文档来挑选特效。）

再增加一点难度：只将特效加到真正的新消息上。当用户初次访问或者刷新浏览器时，不要给之前已经加载过的聊天消息加特效。

如何辨别消息是新的还是旧的？每个消息都有一个数据属性——时间戳，它可以告诉我们该消息是否已经超过了1~2秒。

18.9　中级挑战：缓存消息

将websockets-server.js中的这几行注释掉：

```
    messages.forEach(function (msg) {
      socket.send(msg);
    });
```

那么，在聊天过程中刷新浏览器，所有消息都会消失。之前用UserStore记住了用户名——要是聊天消息也能像用户名一样被记住就更好了。

创建一个MessageStore作为Store的子类。每当消息到达时，它就存储消息，并且保证不会重复存储同一条消息。

页面加载后，Chattrbox应该能从MessageStore中得到任何缓存过的消息。请自己决定是否要在关闭浏览器标签或者重启浏览器之后仍然保存消息缓存。（如果是，应该用什么来替代sessionStorage呢？）

18.10 高级挑战：独立的聊天室

这项挑战需要修改服务端和客户端的应用程序。

为用户添加单独的聊天室。当用户输入用户名后，请求用户输入他们喜欢的聊天室的名称。

当用户登录到一个聊天室，通过WebSocket连接，他们只应该收到来自这一个聊天室的消息。你可能需要改变消息在服务器上存储的方式，或者消息发送到客户端的方式，或者两者都要修改。

再加大点难度：在客户端UI显示一个下拉框，以便用户能切换聊天室。当切换到另一个聊天室时，确保用户能从服务端收到新消息并将其展示到聊天列表中。

第四部分

应用架构

第 19 章

初识MVC和Ember

模型–视图–控制器（Model-View-Controller，MVC）是一种非常有用的软件设计模式。在进行Web应用开发时，可以使用MVC模式分层实现程序结构。本章会详细介绍MVC模式，此外还会介绍如何安装和设置基于MVC模式的框架Ember。之后几章将会分别关注MVC的每一层，逐层完成一款完整应用的开发。

业界对MVC模式有多种解读方式，在前端领域更是如此。图19-1展示了我们选用的解读方式。

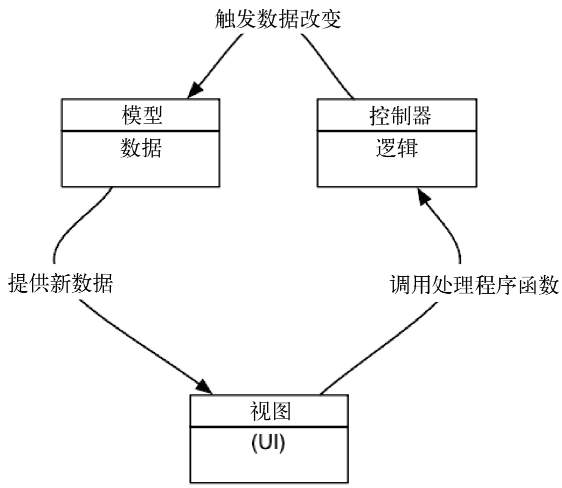


图19-1　MVC模式

各层功能大致如下所示。

- **模型**负责管理数据。当数据发生改变时，模型会通知所有监听者。
- **视图**负责管理用户界面。视图的功能是渲染模型中的数据，同时监听数据变化。此外，它还会在用户触发UI事件时调用控制器中的处理程序函数。
- **控制器**负责应用程序逻辑。它将查询到的模型实例传递给视图。另外，控制器中还包含UI事件的处理程序函数，这些函数可以对模型实例进行变更。

这三部分分工合作，形成环状结构：应用数据从模型流向视图进行渲染，事件数据从视图流

向控制器，控制器则根据UI事件对数据进行修改。

你也许想知道如何走进MVC的大门。在第8章中，我们创建了CoffeeRun。当时，我们将用到的所有模块都按各自的功能进行命名，然后将模块添加到`Window.App`对象中。如果换成MVC模式，则首先需要一个初始化方法，类似于新建一个`Truck`实例，然后在其中依次加载所有的控制器、模型和视图。

从本章开始，我们将开发一款名叫Tracker的新应用。这个应用会以空`<body>`标签的形式在HTML文件中加载DOM的初始状态，此外还会加载一个用于初始化应用的脚本文件。应用会根据当前路由路径和数据状态（模型）动态渲染视图（HTML内容）。

在CoffeeRun应用中，我们共创建了7个模块，而这次会创建更多。在MVC模式统一的组织架构下，可以将所有的模块按功能分割成独立的文件，这样即便有成百上千个模块也能得到很好的管理。

19.1 Tracker

Tracker应用会包含Web应用中最棒的功能之一：URL路由。Tracker还会包含：定义数据的模型、处理用户动作的控制器、定义UI的模板，还有将模型传递给模板的路由。在开发的过程中，还能掌握一些新的模式和技术，它们能让代码更加精练和优雅。

Tracker的目标客户是一群"神秘生物研究者"，他们环游世界，追寻各种神秘生物，如野人、卓柏卡布拉（一种被怀疑存在于美洲的吸血动物）、尼斯湖水怪、独角兽等。他们需要一个应用来跟踪这些神秘生物并对生物的目击记录进行收集。具体来说，目前的需求是下面这些功能点，当然这些需求有可能会发生变更（并且经常变更）。

- 展示所有目击记录。
- 添加新目击记录。
- 将生物与目击记录关联。
- 通过提示消息查看最新一条目击记录。

每个目击记录模型均包含如下属性。

目击记录模型属性	属性类型
出现的日期	`date`对象
出现的地点	字符串
神秘生物	神秘生物主键
目击者	目击者主键组成的数组

每个神秘生物模型均包含如下属性。

神秘生物模型属性	属性类型
名称	字符串
类型	字符串
照片路径	字符串

每个目击者模型均包含如下属性。

目击者模型属性	属性类型
名	字符串
姓	字符串
姓名	字符串：包含姓和名
邮箱	字符串
目击记录	目击记录表主键组成的数组

与开发前面的几款应用不同，这一次开发Tracker的过程会尽量接近真实环境中的应用开发，这也就意味着每一节中的代码会增多，而相应的提示和说明会减少。希望你能从中获得更加真实的应用开发体验，而且能开发出一款令自己满意的复杂应用（如图19-2所示　）。

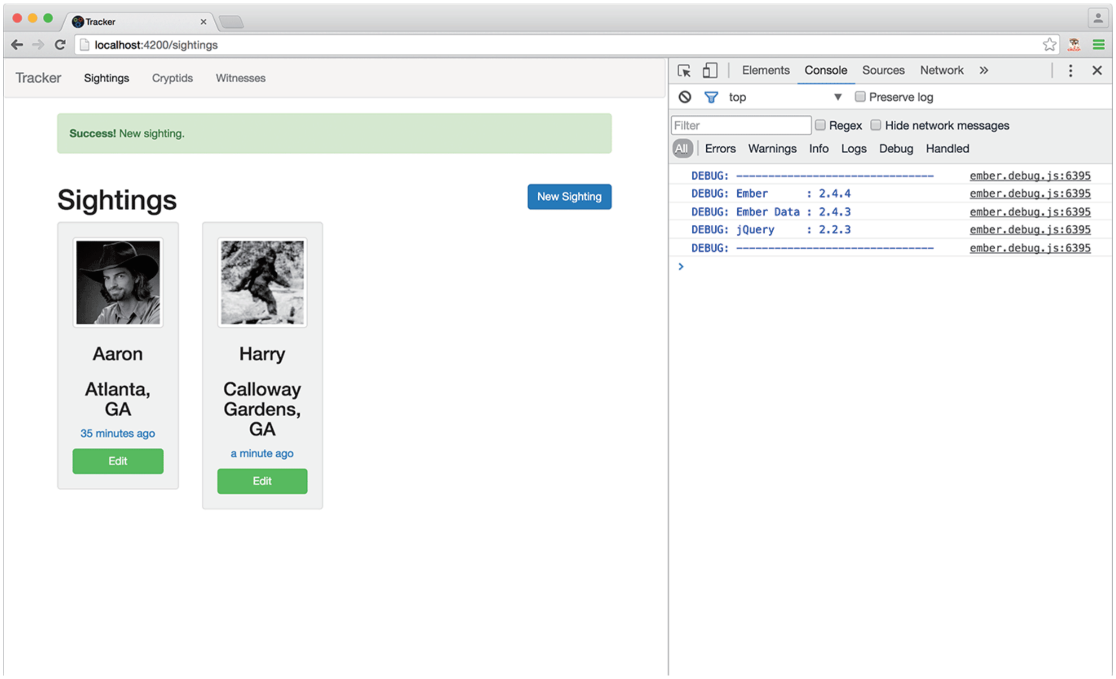


图19-2　完成后的Tracker应用

19.2　Ember：一款 MVC 框架

Ember是一款非常优秀的MVC框架，我们将在开发Tracker的过程中逐步学习如何使用它。为了使开发更加方便迅速，Ember规定了一些概念和命名习惯，在后面的开发过程中会一一介绍。

Ember的官网（emberjs.com）上描述说，Ember是"一款为构建伟大的Web应用而生的框架"。与jQuery等库不同，Ember这种类型的框架通常对项目结构有一定的要求，而且为了快速生成符

合结构的文件，它们多会提供**脚手架工具**——一些用于建立目录和生成样板文件的辅助脚本。为了进一步提升开发者的效率，Ember社区从2011年就开始着手建立一个丰富的生态系统，其中包含了各种库和工具。

Ember的学习之旅将从Ember CLI开始，它就是Ember提供的脚手架工具，包含了开发、测试、构建等各个环节需要的功能。有些人可能不认识CLI，它是Command-Line Interface（命令行接口）的缩写。在后面的开发过程中会借助Ember CLI来新建项目、加载依赖包、生成Ember对象、构建工程、运行应用等。

19.2.1　安装Ember

在正式开始之前，需要安装一些工具。

首先，确保Node.js是最新版本（版本号高于0.12.0）——可以通过`node --version`命令查看版本号。在编写本书时，Node.js的最新版本是5.5.0。[①]（5.5.0与0.12.0在数值上相距甚远，两个版本在功能上也有巨大差别。如果你想了解Node.js的版本号从0.12.0直接跳到4.0.0的原因以及背景，可以查看维基百科上的文章：en.wikipedia.org/wiki/Node.js。）

如果版本过旧，请从nodejs.org网站上下载安装新版的Node.js。

准备好Node.js后，就可以安装Ember CLI了，在终端输入：

```
npm install -g ember-cli@2.4
```

安装需要花费一些时间。[②]如果看到`Please try running this command again as root/Administrator`的错误提示，则表明当前用户权限不足。这时请**不要**直接在命令前添加`sudo`，因为`npm`和`sudo`一起使用可能会带来一些问题。这时可以执行命令`sudo chown -R $USER/usr/local`，然后重新执行安装命令（不带`sudo`）。

在安装过程中还可能遇到包与当前系统不兼容等错误，不过大部分错误的错误提示中都包含了修正方法。如果仍然无法解决，则可以通过搜索引擎求助，有可能需要更新一些现有的程序。此外，Ember CLI也整理了一份常见问题列表，可以访问ember-cli.com/user-guide/#commonissiues查看。

接下来安装Bower，它也是一个资源管理工具。

```
npm install -g bower
```

Bower和npm都是创建Ember应用所必需的。

接下来在Chrome里安装Ember Inspector插件。打开Chrome，在地址栏输入chrome://extensions/，点击页面底部的Get more extensions。在打开的页面中通过搜索找到"Ember Inspector"（如图19-3所示），点击Add to Chrome，然后根据提示完成安装。

① 翻译此书时，Node.js版本已经更新到了7.0.0。——译者注

② 在国内从npm下载包时，某些情况下可能会出现速度极慢或连接超时出错的情况，此时可以使用淘宝提供的npm镜像，详情访问：npm.taobao.org。——译者注

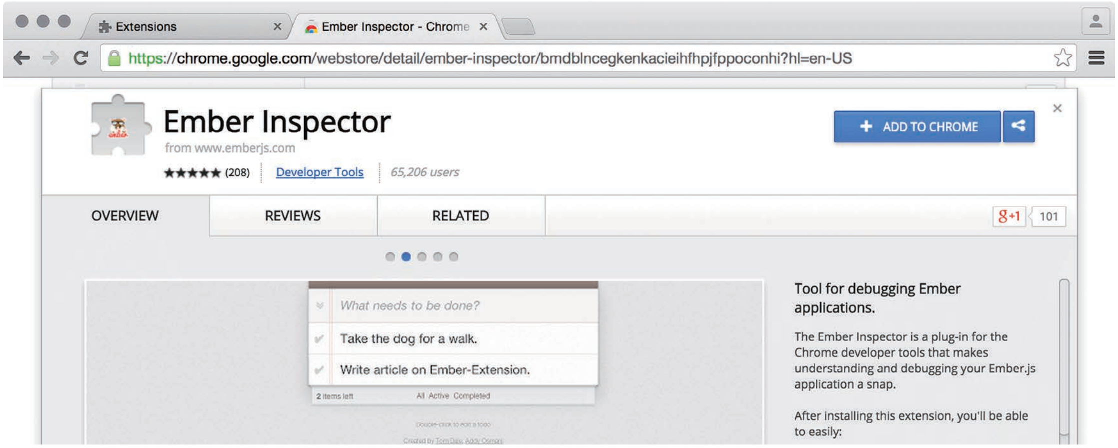


图19-3　在Chrome里安装Ember Inspector插件

Ember CLI在运行过程中需要用到Watchman。Watchman是一款命令行工具，它通过与浏览器通信，实现了无须手动刷新，即可实时更新页面。

如果你用的是Mac电脑，那么可以通过Homebrew安装Watchman。Homebrew是OS X中的一个软件管理工具，安装方法十分简单，只需要从brew.sh网站复制命令然后在终端执行即可。Homebrew安装完成后，就可以使用下面的命令安装Watchman（3.0.0版及以上）了：

```
brew install watchman
```

如果你的系统是Windows，那么请从这里查看安装方法：facebook.github.io/watchman/docs/install.html。

至此，正式开工前的所有准备工作已经全部完成了。

19.2.2　创建Ember应用

通过Ember提供的交互式向导，用寥寥几行代码就可以创建一个应用。在应用启动时，Ember框架会在幕后做许多工作，生成一系列对象和事件。在后面的开发中，我们会逐渐用自己创建的对象替换Ember自动生成的对象。

在执行Ember的`ember new [project name]`命令时，会创建一个新文件夹，并在其中生成项目开始时所必需的文件。

创建一个名叫Tracker的Ember应用。打开终端，切换到项目目录，然后执行：

```
ember new tracker
```

创建Ember应用需要花费一些时间。从终端显示的信息中可以看到，`ember new`命令创建了基本的目录结构和一些项目文件。此外，Ember还通过npm和Bower加载了一些外部库，这些库是Ember应用运行和Ember服务器在执行编译、构建、测试等操作时所必需的。

```
installing app
  create .bowerrc
  create .editorconfig
  create .ember-cli
  create .jshintrc
  create .travis.yml
  create .watchmanconfig
  create README.md
  create app/app.js
  create app/components/.gitkeep
  create app/controllers/.gitkeep
  create app/helpers/.gitkeep
  create app/index.html
  create app/models/.gitkeep
  create app/router.js
  create app/routes/.gitkeep
  create app/styles/app.css
  create app/templates/application.hbs
  . . .
Successfully initialized git.
Installed packages for tooling via npm.
Installed browser packages via Bower.
```


创建过程完成之后，启动本地服务器，检查一下Tracker应用是否正常运行。

19.2.3 启动服务器

接下来执行`ember server`（可以简写为`ember s`）命令，用来编译项目并启动一个可以从本地访问的服务器。`ember server`提供了一项非常方便的功能：它会持续监控本地文件，当发现文件改动时自动进行重新编译、重启服务器等操作，这样在浏览器里看到的永远是最新的代码（它的功能类似于在Ottergram和CoffeeRun中用过的browser-sync）。

Ember CLI使用Broccoli进行编译。JavaScript中的“编译”和Java、Objective-C等语言中的“编译”可能有所不同，前者是指通过分析JavaScript文件的依赖关系，将运行应用所需要的全部文件合并到一起，此外还会加载所有的依赖。

接下来就要“为用户而战”了。请切换到Tracker项目的目录，然后启动服务器：

```
cd tracker
ember server
```

打开Chrome，新建一个标签访问http://localhost:4200，就能看到刚才创建的应用。接着打开开发者工具，切换到Ember标签页，能看到前面安装的调试工具（如图19-4所示）。

前面提到过，Ember CLI会在检测到文件变化时自动刷新浏览器的页面，这个功能叫作**实时加载**（`Livereload`），在终端中也能看到相关的提示：

```
Livereload server on http://localhost:49152
```

如图19-4所示，在控制台和Ember Inspector中都列出了生成的组件和它们的版本号。本书中使用的Ember和Ember Data版本号均为2.4。本书出版时，Ember CLI自动下载的框架版本是2.*x*，

如果你在开发者工具中看到的版本号仍旧是1.*x*.*x*，说明你可能跳过了前面安装与升级Ember CLI的部分。

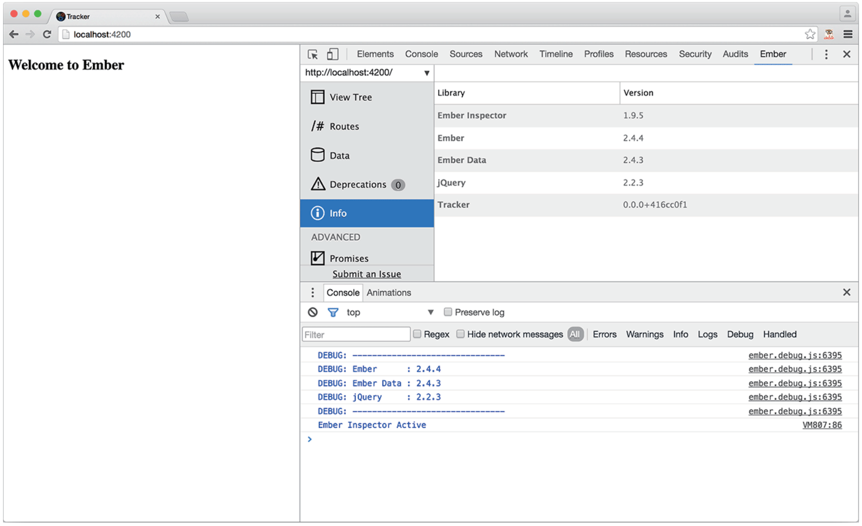


图19-4　Ember服务器

（请注意，在终端中启动Ember服务器后显示的是Ember CLI的版本号，而不是Ember框架的版本号。）

19.3　安装外部库和插件

Ember CLI为开发者提供了许多加速开发的途径，其中包括引入一些开源代码。在前几章中，你使用过npm向本地环境安装node模块，本章前面也讨论过通过另一个包管理器（Bower）加载外部库。

Ember CLI能够很好地与这两个包管理器搭配使用。通过它们安装外部库或工具的命令如下：

```
npm install [package name] --save-dev
npm install [package name] --save
bower install [package name] --save
```

执行这些命令时，包管理器会自动下载外部库的文件，并保存到bower_components或node_modules目录中。

在CoffeeRun项目中用过的Bootstrap，在这次的Tracker中还会用到。要使用Bower安装Bootstrap，输入以下命令即可：

```
bower install bootstrap-sass --save
```

现在Bootstrap库已经下载到了本地，包括其中的JavaScript文件和样式文件。下一步要做的是把它们添加到Ember CLI的构建过程中，只有这样才能在应用中访问Bootstrap提供的资源。

在现代前端工作流中，无论是开发脚本还是样式，都逃不掉繁杂的编译工作。这里借助工具ember-cli-sass可以降低编译的复杂性，它会在Ember CLI整体的编译流程中添加一个步骤：将SCSS文件转化为CSS文件。SCSS，有时也称为Sass，在不改变我们熟知且喜爱的CSS语法的前提下，为其增加了许多常用的逻辑功能，例如变量、函数、循环、键值对等。

在终端中安装ember-cli-sass：

```
ember install ember-cli-sass
```

ember-cli-sass只是Ember众多插件中的一个。Ember有一个插件库（www.emberaddons.com），其中包含了许多外部库和配置代码、创建好的辅助方法和组件以及其他类型的辅助工具，这些插件可以通过Ember CLI的命令`ember install`进行安装。

注意，Ember CLI是一个相对较新的工具，而某些插件有可能已经过时。如果在安装插件时遇到问题，可以去这个插件的GitHub问题页面（Issue）看看。

刚才已经为应用添加了编译SCSS的功能，现在尝试把项目中的.css文件修改为.scss文件：将app/styles/app.css重命名为app/styles/app.scss。然后重启Ember服务器以便重新加载文件。

现在来测试一下：在样式表中添加一个SCSS变量。打开app/styles/app.scss文件，以键值对的形式添加一个变量（注意，变量名必须以$开头）：


```
$bg-color: coral;
html {
  background: $bg-color;
}
```

查看浏览器，应该就能看到页面多了背景色（如图19-5所示）。

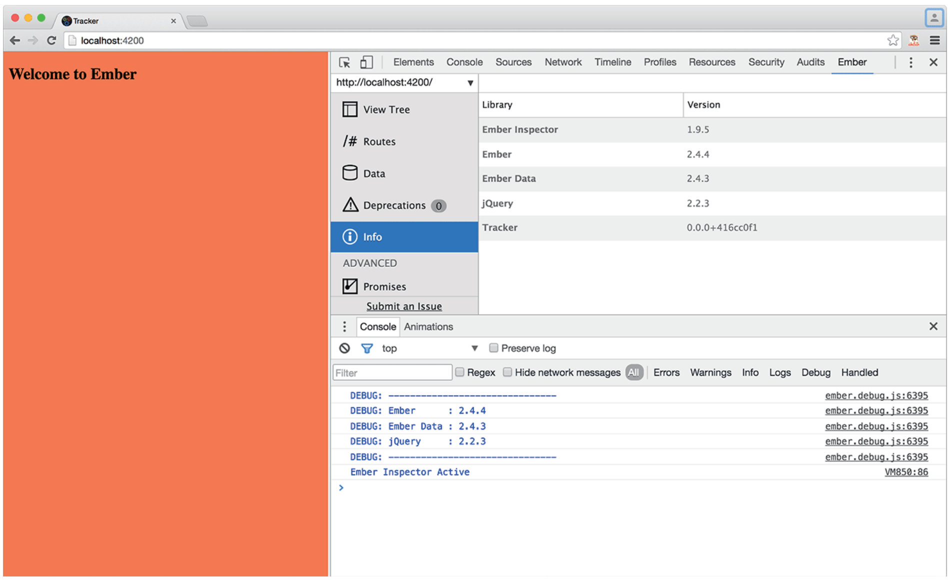


图19-5　测试编译SCSS

下一步是将Bootstrap的样式和脚本引入到项目中。之前通过`bower install bootstrap-sass`命令安装了SCSS版本的Bootstrap，现在需要将它添加到项目中。首先在样式表中引入这个库，然后修改Ember CLI的配置文件以便在编译时加载它。

19.4　修改配置

Broccoli就是前面提到过的编译引擎。新增JavaScript和样式文件时，需要对它的配置做一些修改。

配置文件叫ember-cli-build.js，由Ember CLI生成。不论是向项目中添加依赖，还是修改应用的输出结构，都可以通过修改这个配置文件实现。对Tracker应用来说，只需要添加外部依赖库以及修改SCSS的编译配置。

打开ember-cli-build.js文件。首先添加一个变量，指向Bootstrap资源目录的路径；接着在配置中添加一个`sassOptions`对象，它只包含一个属性`includePaths`，值为Bootstrap样式表的路径：

```
...
var EmberApp = require('ember-cli/lib/broccoli/ember-app');

module.exports = function(defaults) {

  var bootstrapPath = 'bower_components/bootstrap-sass/assets/';

  var app = new EmberApp(defaults, {
    // 此处加入选项
    sassOptions: {
      includePaths: [
        bootstrapPath + 'stylesheets'
      ]
    }
  });

  // ……模板注释……

  // 生成指向bootstrap资源文件的路径
  // 使用import向应用添加对该资源的引用
  app.import(bootstrapPath + 'javascripts/bootstrap.js');

  return app.toTree();
};
```

有了以上配置，Ember CLI就可以从bower_components/bootstrap-sass/assets/stylesheets目录中加载*.scss文件进行编译了。保存修改后的配置文件，重启Ember服务器使新的配置生效。

接下来，在app.scss中使用`@import`指令，引入Bootstrap的样式：

```
$bg-color: coral;
html {
  background: $bg-color;
}
```

```
// ---------------------------
// bootstrap variable overrides
// ---------------------------

// end bootstrap variable overrides
@import 'bootstrap';
```

`@import`指令的功能是将bootstrap.scss文件的内容添加到app.scss中，这个过程将由Ember CLI的构建过程代劳。Bootstrap的样式文件所在的目录是bower_components/bootstrap-sass/assets/stylesheets/。

你还在配置文件中添加了`app.import(bootstrapPath + 'javascripts/bootstrap.js');`，用于在Ember CLI构建时引入Bootstrap的JavaScript组件。引入的所有文件都会被添加到一个列表中，最后被拼接成一个文件/dist/assets/vendor.js。对Bootstrap来说，bootstrap.js中包含了各个独立的模块，例如`collapse`、`modal`、`tab`、`dropdown`等。将所有这些模块都添加到项目中可能会有些臃肿，所以后面你可以调整配置，选择性地添加项目会用到的模块。

资源文件添加完成后，先来确认一下它是否生效，再进行下一步。这时你可能会想到应用目录中的index.html文件，但最好不要在这里测试代码，这个文件主要是为构建过程服务的。

正确的做法是将HTML元素添加到应用**模板**中，模板存放在在app/templates目录中。关于模板的细节会在第23章介绍。

现在，打开app/templates/application.hbs文件，添加一个Bootstrap的NavBar组件：

```
<h2 id="title">Welcome to Ember</h2>

{{outlet}}
<header>
  <nav class="navbar navbar-default">
    <div class="container-fluid">
      <!--将APP名称和展开按钮放到一个层中，以便进行移动端适配-->
      <div class="navbar-header">
        <button type="button" class="navbar-toggle collapsed"
          data-toggle="collapse" data-target="#top-navbar-collapse">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand">Tracker</a>
      </div>

      <!--将导航链接、表单和其他内容放到弹出层中-->
      <div class="collapse navbar-collapse" id="top-navbar-collapse">
        <ul class="nav navbar-nav">
          <li>
            <a href="#">Test Link</a>
          </li>
          <li>
            <a href="#">Test Link</a>
          </li>
```

```
          </ul>
        </div><!-- /.navbar-collapse -->
      </div><!-- /.container-fluid -->
    </nav>
  </header>
  <div class="container">
    {{outlet}}
  </div>
```

这里添加了NavBar组件以及相应的HTML属性，如`id`、`class`、`name`、`data`等。另外，还把模板中原有的`{{outlet}}`移到了`<div>`标签中。`{{outlet}}`在模板文件中代表子模板，下一章会详细介绍。

图19-6展示了这段代码的效果。

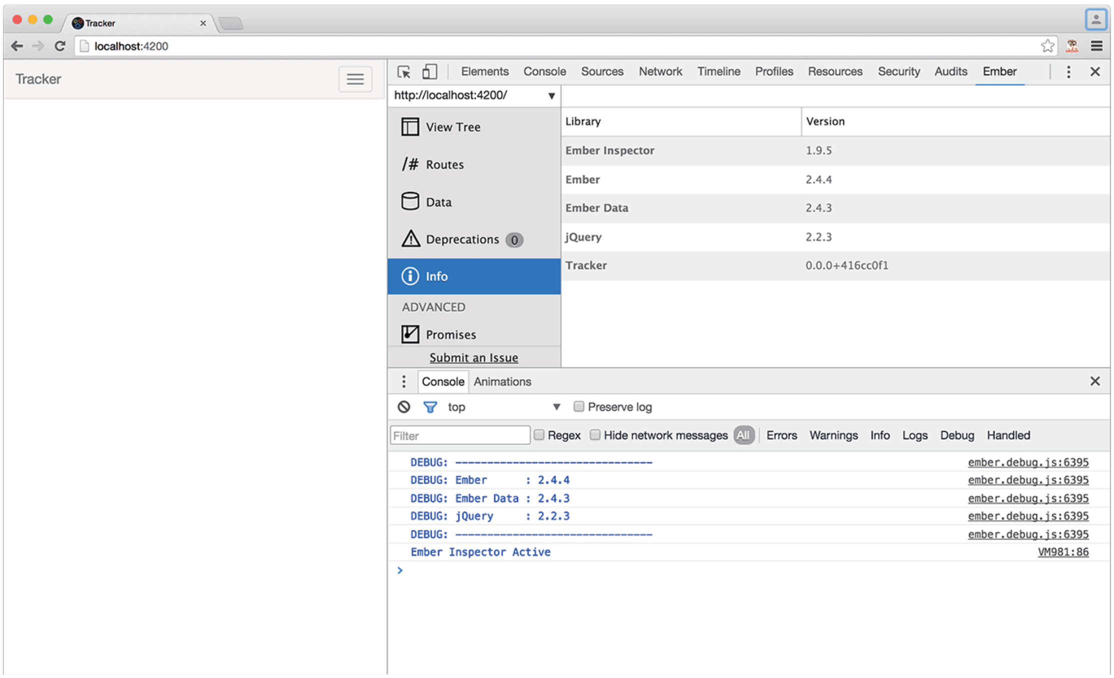


图19-6　Bootstrap NavBar

NavBar支持响应式——当窗口宽度小于768px时，菜单会自动收起，默认只展示一个按钮。点击按钮可以显示或隐藏菜单（如图19-7所示）。这个事件的监听和处理由bootstrap.js完成。

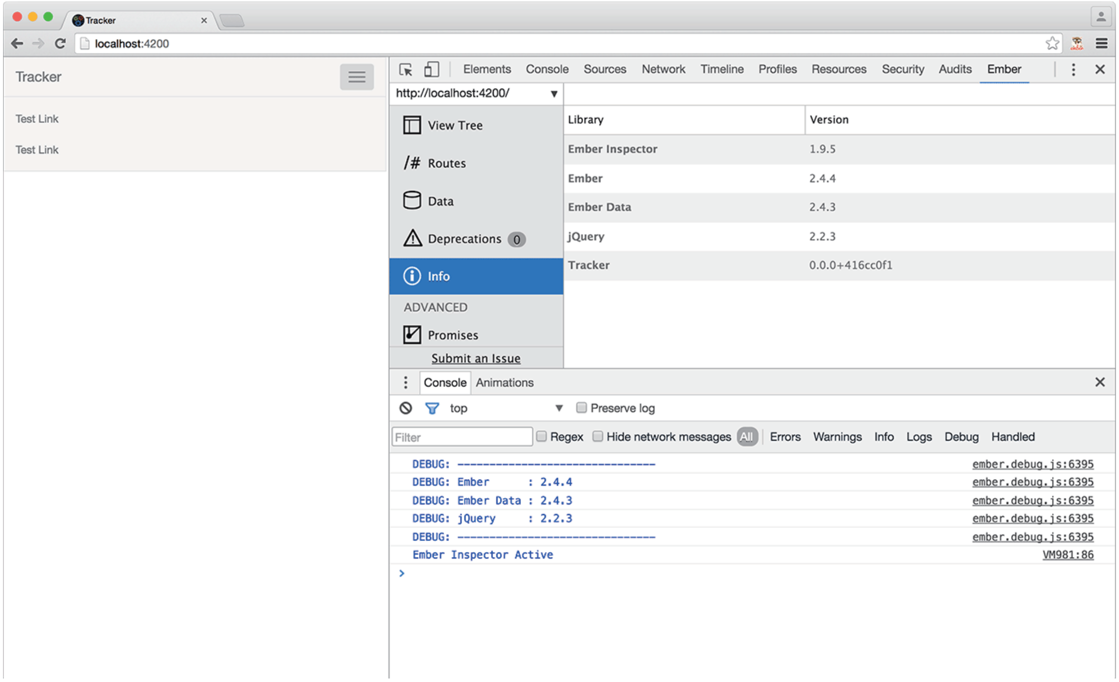


图19-7　测试Bootstrap NavBar组件的展开与折叠

恭喜！你的第一个Ember应用已经可以运行了。在整个过程中，你通过安装的工具完成了生成代码、编译资源文件、加载依赖以及架设服务器等操作。你已经为这个应用打下了坚实的基础，后面几章将完成剩下的开发工作。

19.5　延展阅读：npm 和 Bower 的安装命令

`npm install`和`bower install`命令后的`--save-dev`和`--save`参数会在它们各自的配置文件中添加一个键值对，用来描述安装的模块的名称和版本。Bower的配置文件是bower.json，npm的配置文件是package.json（在Chattrbox中出现过）。

以下是bower.json的一个示例，可以看到新添加的键值对：

```
{
  "name": "tracker",
  "dependencies": {
    "ember": "~2.4.3",
    "ember-cli-shims": "0.1.1",
    "ember-cli-test-loader": "0.2.2",
    "ember-qunit-notifications": "0.1.0",
    "bootstrap-sass": "^3.3.6"
  }
}
```

bower.json中列出了所依赖的ember.js库以及要求的最低版本号。代码仓库或版本控制系统中并不会保存这些资源，它们只保存bower.json文件。开发者在检出代码后可以通过`bower install`和`npm install`下载这些资源，完成环境的搭建。

19.6　初级挑战：限制引入

请修改ember-cli-build.js文件，删除对bootstrap.js整体的引入，只选择性地引入collapse.js和transition.js两个文件。这样可以在不影响NavBar运行的前提下，缩小编译生成的vendor.js文件。

在动手修改前，找到dist/assets/vendor.js文件，记录下它的行数（或者文件大小）并和修改后的行数做个对比。

19.7　中级挑战：添加 Font Awesome 库

Font Awesome是一款UI库，它提供了一系列常用的图标，而且这些图标像文字一样可以自由缩放。请从Ember CLI的插件库引入Font Awesome，并为app/templates/application.hbs添加一个图标。可以在Font Awesome的GitHub主页上查看更多信息。

19.8　高级挑战：自定义 NavBar

Bootstrap的样式是以SCSS编写的，其中使用了许多灵活的变量和函数。如果在项目中引用的是SCSS版本的Bootstrap，那么你就可以很方便地控制它的样式规则——甚至还可以新建一套样式，通过修改变量覆盖Bootstrap提供的默认值。

请尝试仅通过在app/stylesheets/app.scss文件中添加或修改变量，对NavBar的`background-color`、`border-radius`和`padding`进行修改。

第 20 章

路由选择、路由表、模型

完成上一章后，Tracker应用已经拥有了基本的框架，现在需要为它填充页面以及定制**路由规则**。

路由选择就像交警指挥交通，它会根据用户输入的URL选择要渲染的页面。在之前的几个项目中，你曾为表单提交和按钮点击定义了事件监听器。路由选择和事件监听器有些相似，不过它监听当前URL的变动。

每个网站都会用到路由选择，虽然具体方案可能不尽相同。例如，访问一个URL：www.bignerdranch.com/we-teach/。服务器会根据路由/we-teach/找到对应的文件夹we-teach，继而找到并渲染其中的HTML文件。对有些网站来说可能是另外一种情形：服务器并不会去寻找静态的HTML文件，而是运行一段程序，动态生成HTML代码。

Ember应用可以实现类似的功能，但它不需要向服务器请求HTML。当需要进行页面切换时，Ember首先把应用当前的地址修改为目的页面的地址。这时路由模块`Router`（应用核心对象的一个子元素，包含对URL改变事件的监听器和处理程序）会根据新的地址查询路由表，找到对应的路由对象（`Ember.Route`）。接着，它调用路由对象中的一系列回调方法，开始为目的页面准备数据，这一系列回调方法被叫作**路由生命周期钩子**（route lifecycle hook）。

创建路由是Ember开发中的一项基本操作。按照Ember的命名规则，控制器和模板需要与路由的名称相匹配。举个例子，假如创建了一个路由sightings，路由模块会将所有对/sightings的请求映射到路由SightingsRoute上，接着执行名为SightingsController的控制器，最后渲染app/templates/sightings.hbs模板。

通过本章的学习，你可以了解Ember应用架构，并且学会如何使Ember CLI创建路由模块和模板文件。路由是Ember应用的关键，学好本章的内容将为后面5章的应用开发做好铺垫。

图20-1展示了本章结束时Tracker的界面。

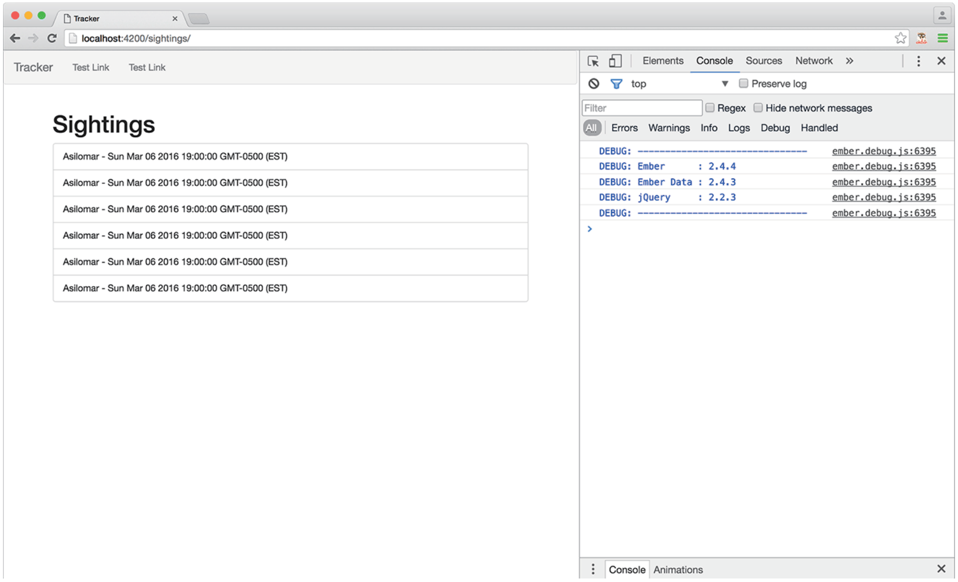


图20-1　Tracker应用

20.1　Ember 生成器

Ember CLI提供了一个生成器的脚手架工具`generate`，它对学习Ember的约定和命名模式很有帮助。通过命令`ember generate`（可以简写为`ember g`）可以生成相应的文件和样板代码。

回想一下，开发Tracker应用的目的是收集神秘生物（例如野人）的目击记录。应用需要收集的信息有：目击记录、神秘生物类型和目击者，所以需要以下路由：

路由名称	路由路径	页面内容
index	/index	无，重定向至 sightings
sightings	/sightings	目击记录列表
cryptids	/cryptids	神秘生物列表
witnesses	/witnesses	目击者列表
sighting	/sighting	目击记录详情
cryptid	/cryptid	神秘生物详情
witness	/witness	目击者详情
sightings index	/sightings/index	目击记录默认页面
sightings new	/sightings/new	创建目击记录的表单
sighting index	/sighting/:sighting_id/index	单条目击记录详情页面
sighting edit	/sighting/:sighting_id/edit	编辑目击记录的表单

现在通过`ember generate`命令创建这些路由。打开终端，进入到tracker目录，依次执行以下命令（每次一行）创建路由：

```
ember g route index
ember g route sightings
ember g route sightings/index
ember g route sightings/new
ember g route sighting
ember g route sighting/index
ember g route sighting/edit
ember g route cryptids
ember g route cryptid
ember g route witnesses
ember g route witness
```

创建的过程如图20-2所示。

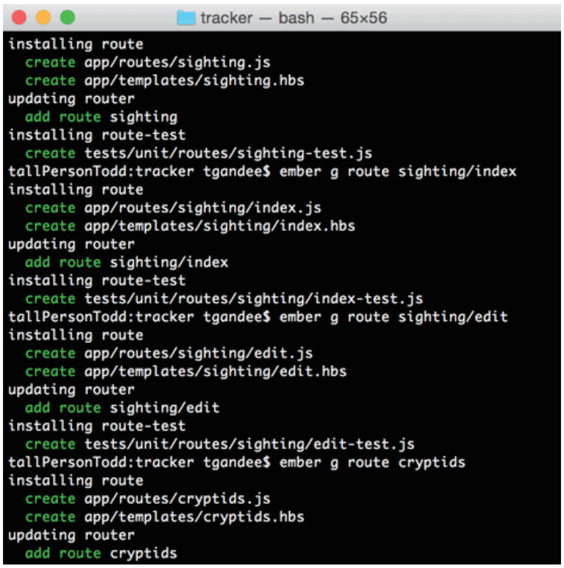


图20-2　创建路由

现在看一下`ember g`命令做了哪些工作。首先，它在routes/和templates/目录下创建了一些文件。

打开app/routes/index.js，能看到在这个模块中引入（`import`）了Ember，最后导出（`export`）了Ember.Route：

```
import Ember from 'ember';

export default Ember.Route.extend({
});
```

`.extend`方法接受一个JavaScript对象作为参数，用以创建一个Ember.Route的子类。使用ES6的模块语法可以为每条路由都创建独立的模块。

无论是使用生成器创建路由（这里选用的做法）还是手动创建路由，Ember CLI都能自动找到Ember.Route，并将其加载到项目中。当然，使用生成器会更加方便，因为它在创建文件的同时会添加样板代码。

打开app/templates/index.hbs（它是Ember CLI为`IndexRoute`生成的模板文件），能看到其中只有一行代码：`{{outlet}}`。

你应该有印象，在上一章中也出现过这句代码，是在templates/application.hbs文件中。它的作用是在路由层级之间实现内容嵌套，稍后就会对它进行详细讲解。

暂时先不动`{{outlet}}`，只在它前面添加一个`<h1>`元素：

```
<h1>Index Route</h1>
{{outlet}}
```

运行`ember server`启动服务器，开发过程中让它保持后台运行就好。如果同时还需要运行其他Ember CLI命令（例如调用生成器），则新开一个终端窗口。在加入新模块时，服务器在运行中会自动将其加载到应用中，并会刷新浏览器。

打开Chrome，访问http://localhost:4200，应该能看到如图20-3所示的页面。

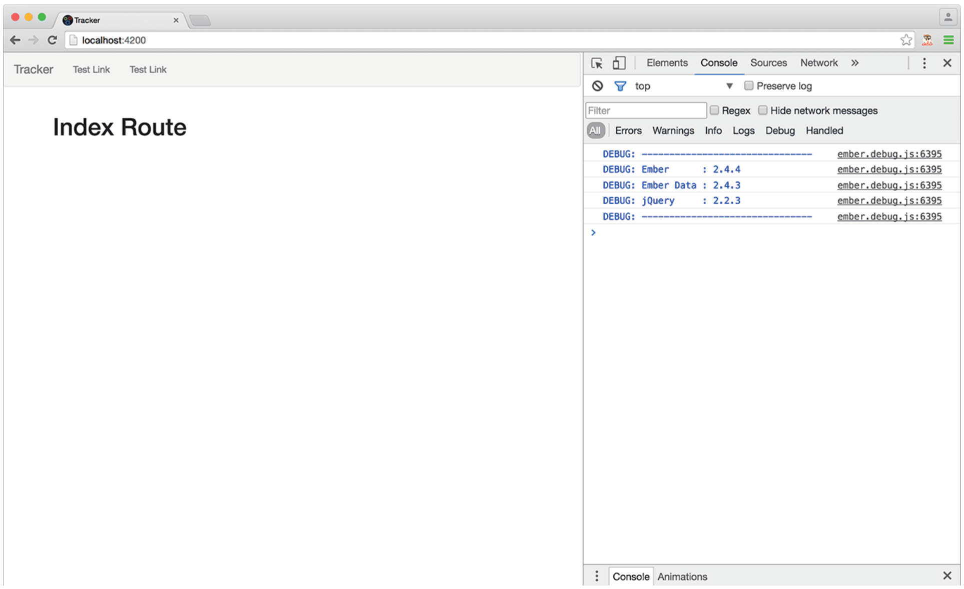


图20-3　首页

在这个页面中，能看到app/templates/application.hbs中的NavBar组件和app/templates/index.hbs中的`<h1>`元素。它们为什么会在这里一起出现？

在创建应用的过程中，Ember自动生成了许多文件，其中包括app.js和router.js。app.js是应用的入口，用于处理初始化之类的工作。其中包含一些逻辑，用于在初始化时创建一个Ember的实例，这个过程与在CoffeeRun中创建`Truck`类似。

Ember应用在启动或重启时会对路由模块（Router）和应用路由（ApplicationRoute）进行实例化。这两个对象十分关键，它们控制着整个应用。

在router.js中，通过注册路由，可以将URL绑定到特定的页面上。在注册路由时可以提供一些配置参数，还可以创建嵌套结构的路由。这也是Ember一项非常强大的功能：能够在不同的页面中复用逻辑和内容。

打开router.js，查看其中注册路由的方法：

```
import Ember from 'ember';
import config from './config/environment';

const Router = Ember.Router.extend({
  location: config.locationType
});

Router.map(function() {
  this.route('sightings', function() {
    this.route('new');
  });
  this.route('sighting', function() {
    this.route('edit');
  });
  this.route('cryptids');
  this.route('cryptid');
  this.route('witnesses');
  this.route('witness');
});

export default Router;
```


这段代码中的Router.map接受一个回调函数作为参数。在这个回调函数中，使用route方法注册路由。还可以为route方法传入回调函数作为第二个参数以实现路由的嵌套。Ember会将这些嵌套转换成相应的路由层级，在层级的最顶端是ApplicationRoute。

当用户访问一个映射到嵌套路由的URL时，Ember会首先调取父级路由对应的模板。在其中寻找{{outlet}}标签，这个标签代表着“这里要替换成子模板”。

来看看这个逻辑在实际应用中的效果。

在应用首页的页面中，首页路由（IndexRoute）对应的模板嵌套在应用路由（ApplicationRoute）的模板中。这背后还隐藏了一个逻辑，Ember会自动在项目的routes/文件夹下寻找名为index.js的文件。对于任意一条路由，都可以在它对应的路由文件夹下创建一个index.js文件，作为该路由的默认页面。这是一种通用做法，在其他Ember应用中也应该这样操作。

不知道你有没有注意到，router.js中并没有与index相关的路由。事实上，Ember会为所有嵌套结构中的父级路由自动绑定一条index子路由，就像在router.js中定义的一样：

```
...
Router.map(function() {
  this.route('index');
  this.route('sightings', function() {
```

```
      this.route('index');
      this.route('new');
    });
    this.route('sighting', function() {
      this.route('index');
      this.route('edit');
    });
    this.route('cryptids');
    this.route('cryptid');
    this.route('witnesses');
    this.route('witness');
  });
  ...
```

20.2 嵌套路由

通过路由，可以很方便地控制视图中展示的内容。就像文件夹一样，嵌套的路由将基于URL的一系列相关路由聚合在一起。可以将父级路由理解成名词，将子路由理解成动词或形容词：

```
//父级路由就像名词
this.route('sightings', function() {
  //子路由就像动词或者形容词
  this.route('new');
});
```

这里sightings是一个父级路由，代表着**目击记录列表**（名词），而其中嵌套的子路由new代表着**创建**一条目击记录（动词）。无论是父级路由还是子路由，都通过`this.route`绑定到URL。

通过模板嵌套，可以让一部分内容在所有页面上都显示（例如导航条），而其他内容都只在特定的页面中显示（例如首页路由的模板只在首页中显示）。需要用每条路由的回调函数指定它获取数据的方法。

现在，对添加路由时自动生成的一系列模板进行一些小修改，然后查看各个页面修改后的效果。在这一节中添加的所有代码都只是临时性的，目的是帮助你理解路由间的关系。

首先修改app/templates/sightings.hbs模板，在`{{outlet}}`标签的上方添加一个`<h1>`元素。

```
<h1>Sigophtings</h1>
{{outlet}}
```

接下来编辑app/templates/sightings/index.hbs模板。这一次将其中的`{{outlet}}`标签直接替换成`<h1>`。在父级模板中，`{{outlet}}`标签代表被嵌套的子模板。而对于app/templates/sightings/index.hbs来说，它本身已经是最末级路由的模板，不存在子模板，所以不再需要`{{outlet}}`。

```
{{outlet}}
<h1>Index Route</h1>
```

保存文件，访问http://localhost:4200/sightings/查看效果（如图20-4所示）。

接下来，编辑app/templates/sightings/new.hbs。这个路由同样也是最末级，所以也用同样的方式处理：删掉`{{outlet}}`，替换成`<h1>`。

```
{{outlet}}
<h1>New Route</h1>
```

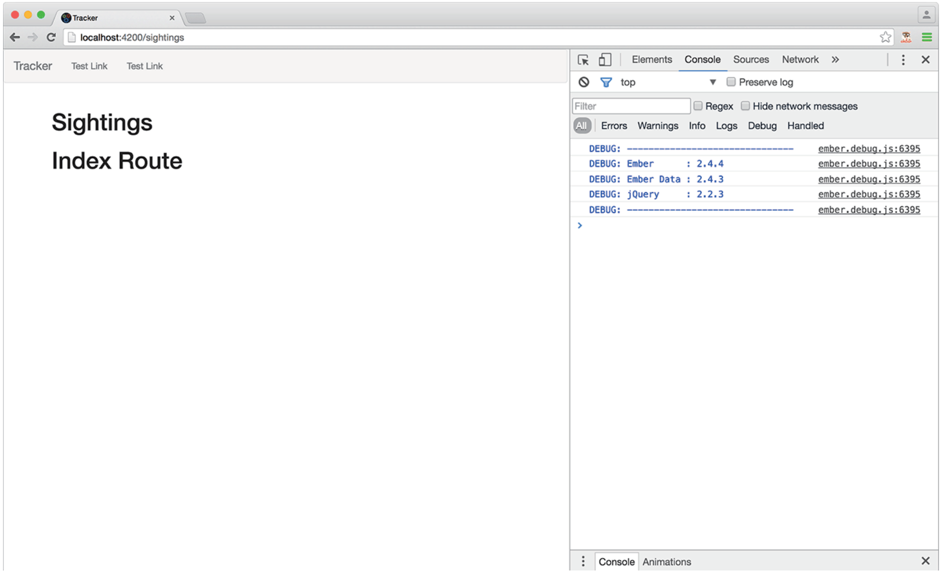


图20-4　目击记录：嵌套路由

现在访问http://localhost:4200/sightings/new（如图20-5所示）。

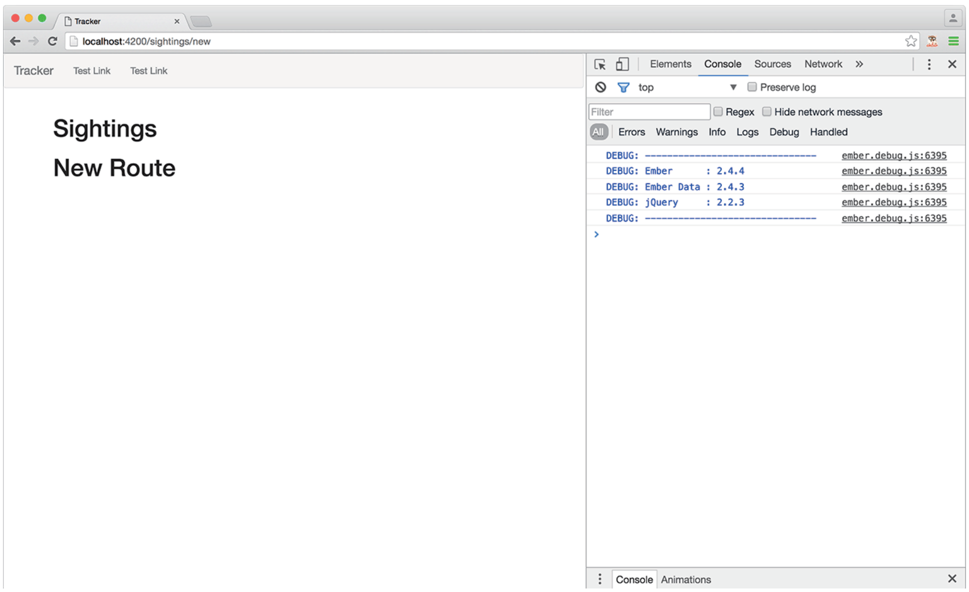


图20-5　目击记录：new路由

现在Tracker应用的路由已经拥有了嵌套结构，包含父模板app/templates/sightings.hbs，以及在其中使用`{{outlet}}`标签渲染的子模板（例如app/templates/sightings/index.hbs和app/templates/sightings/new.hbs）。

20.3　Ember Inspector

Ember Inspector提供了一种查看应用中的全部路由的便捷途径。在Ember Inspector中点击Routes菜单，如图20-6所示。

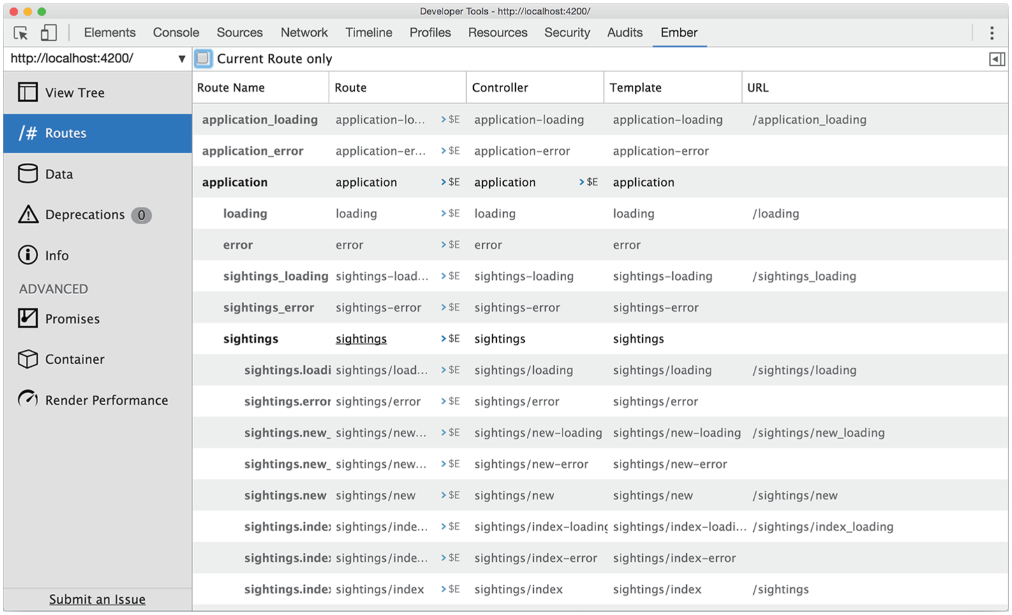


图20-6　路由结构

好多路由！这里展示的路由比实际创建的多太多了。仔细观察能发现，其中许多路由都以`loading`或`error`结尾。事实上这些路由和`index`一样都是由Ember自动创建的，为数据加载过程中的各个状态服务，目的是补全路由在不同状态间切换时的空隙。

20.4　指派模型

接下来要做的是使用路由对象的`model`方法获取数据。每个`Ember.Route`对象都有一个`model`方法，它的作用是将模型（还记得模型吗？模型就是支撑模板的数据）指派给控制器，它会以`Promise`对象的形式返回数据。

每当URL发生改变时，应用都会在幕后重新初始化路由对象。路由对象上包含四个**钩子**，可以用来对路由进行一些设置：`beforeModel`、`model`、`afterModel`和`setController`。

首先来看model。

打开app/routes/sightings.js文件，在model方法中添加一些模拟数据：

```
import Ember from 'ember';

export default Ember.Route.extend({
  model(){
    return [
      {
        id: 1,
        location: 'Asilomar',
        sightedAt: new Date('2016-03-07')
      },
      {
        id: 2,
        location: 'Asilomar',
        sightedAt: new Date('2016-03-07')
      },
      {
        id: 3,
        location: 'Asilomar',
        sightedAt: new Date('2016-03-07')
      },
      {
        id: 4,
        location: 'Asilomar',
        sightedAt: new Date('2016-03-07')
      },
      {
        id: 5,
        location: 'Asilomar',
        sightedAt: new Date('2016-03-07')
      },
      {
        id: 6,
        location: 'Asilomar',
        sightedAt: new Date('2016-03-07')
      }
    ];
  }
});
```

注意这里定义model时的语法：

```
model() {
  [你的代码]
}
```

这是ES6中定义的语法，是下面形式的简写：

```
model: function() {
  [你的代码]
}
```

后面的章节都会采用简写形式为对象定义方法。

model钩子可以用来获取渲染模板所需要的数据。Ember.Route对象的每个生命周期方法都会给钩子函数返回一些对象。从model中返回的数据最终会进入setupController钩子，用于将数据赋给SightingsController的model属性。这些数据可以在app/templates/sightings.hbs模板和app/templates/sightings/index.hbs模板中访问。

将app/templates/sightings/index.hbs替换成以下代码，稍后会详细解释：

```
<h1>Index Route</h1>
<div class="panel panel-default">
  <ul class="list-group">
    {{#each model as |sighting|}}
      <li class="list-group-item">
        {{sighting.location}} - {{sighting.sightedAt}}
      </li>
    {{/each}}
  </ul>
</div>
```

如果你以前没有接触过模板语言的话，初看这段代码可能感觉比较奇怪。那些被双大括号{{ }}包起来的代码看上去是语句，但实际上是JavaScript的函数。将这段代码翻译成自然语言，则是“为model属性（类型是数组）中的每一条目击记录（sighting）渲染一个<li>容器，在容器中展示目击记录发生的时间（sightedAt）和地点（location）”。

第23章将会介绍{{ }}语法，还会对{{#each}}进行重点介绍。

访问http://localhost:4200/sightings，应该能看到和图20-7相似的页面。

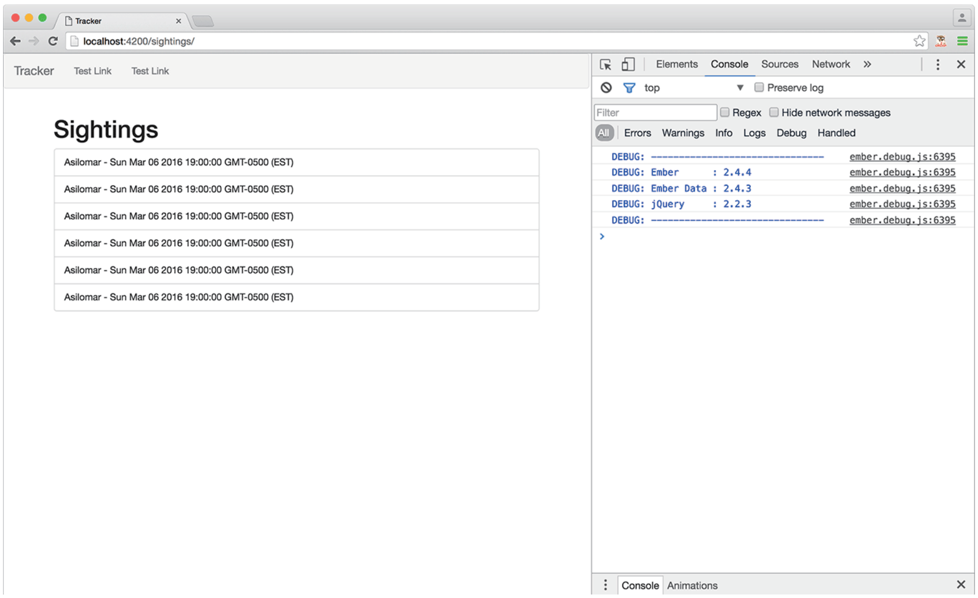


图20-7　首页数据列表

现在已经完成了路由循环中的第一部分，也就是将数据传递给模板进行渲染。第23章将会介绍模板语言Handlebars。借助模板语言，可以通过控制器的属性控制应用的状态，而且只在状态发生改变时渲染必要的DOM元素。

20.5　beforeModel

前面提到过，路由对象会依次调用一系列钩子方法，其中的第一个是beforeModel。这个钩子非常适合用来在请求数据前检查应用的状态，也可以用来校验用户权限，将没有访问权限的用户重定向到其他页面。

现在尝试在首页路由的beforeModel钩子中对用户进行重定向。因为我们的应用目前还没有首页（有兴趣的话你可以创建一个仪表台页面作为首页），所以直接将访问应用首页的用户重定向到目击记录列表页。

打开app/routes/index.js，在其中添加beforeModel钩子：

```
import Ember from 'ember';

export default Ember.Route.extend({
  beforeModel(){
    this.transitionTo('sightings');
  }
});
```

现在，当访问http://localhost:4200/时，浏览器会自动跳转到http://localhost:4200/sightings，也就是由app/templates/sightings/index.hbs模板渲染出的目击记录列表页面。

还剩afterModel和setupController这两个钩子没有介绍，因为在Tracker中并不会用到它们。建立一个路由文件，其实就是创建了Ember.Route对象的副本，并覆盖其中的一些方法，这种做法特别像使用Java或类似语言中的接口。setupController钩子函数会被默认执行，用来将model属性赋值给路由对象的控制器。

到目前为止，Tracker已经具备了基本的路由，这些路由能够大致体现本应用的功能：首页、目击记录列表、添加目击记录的路由。这一章已经为路由创建了模板，还为目击记录的路由对象添加了数据，最后将首页重定向到了目击记录列表页。我们成功地开了一个好头！

下一章会介绍Ember的模型（Ember.Model）、适配器、计算属性以及存储机制。

20.6　延展阅读：setupController 和 afterModel

setupController钩子的用途是在控制器中添加属性供模板使用。执行this._super，就可以在给controller设置其他属性的同时，执行默认行为给controller设置model属性。[①]

① 如果没有覆盖setupController，它会执行默认行为，即为控制器添加model属性，值是从模型获取的数据；如果覆盖了setupController，则默认行为将不会被执行。如果此时还希望使用model属性，则需要显示地调用this._super。——译者注

```
setupController(controller, model) {
  this._super(controller, model);
  // this.controllerFor('[other controller]').set("[property name]", [value]);
}
```

afterModel钩子会在model钩子返回的Promise对象被resolve时执行。注意，在一些特殊情况下，model钩子可能不会被调用，因为在调用它之前Promise就已经被resolve了。在这种情况下，由于afterModel会在setupController之前被调用，所以可以在将模型数据传给控制器之前用afterModel来校验数据的完整性。

第 21 章

模型和数据绑定

本章主要聚焦在应用的数据层。

到目前为止，你已经和对象字面量形式的数据打过很多交道了。前面已经介绍过如何创建和修改对象和对象属性，以及如何用函数快速生成带默认值的对象。另外还介绍了在`localStorage`和`sessionStorage`中存储数据的方法。

在Tracker应用中，你还会学到怎样以模型的形式使用数据。模型的本质就是可以创建包含特殊属性和方法的对象的函数。贯穿在应用中的数据有了模型，便有了结构。

`Ember.Object`是应用中最基础的数据结构，Ember中其他所有类都继承于它。在应用运行期间，借助`Ember.Object`就可以用非常简单的定义和命名模式完成对模型实例的的创建、检索、升级或销毁。

当然，`Ember.Object`提供的功能还远不能满足一款现代应用的需求。在实际应用中，通常需要对模型数据进行持久化存储，以满足业务逻辑中对数据检索或保存的需求。

Ember Data是基于`Ember.Object`构建的JavaScript库，用于开发模型相关的功能。Ember Data提供了一些基于`Ember.Object`构建的类，这些类对各种复杂的数据源（包括RESTful API、localStorage、静态数据等）进行了抽象。

Ember Data同时也提供了基于内存的存储`store`，后面所有对数据的增删改查都通过`store`进行操作。

21.1 定义模型

Ember CLI已经加载了Ember Data库，所以可以直接开始创建模型。

上一章使用Ember的生成器`ember g route` [路由名称]创建了路由和相关文件，这里则使用同样的方法创建模型，命令是`ember g model` [模型名称]。

分别创建路由所需的神秘生物、目击记录和目击者模型：

```
ember g model cryptid
ember g model sighting
ember g model witness
```

神秘生物模型定义在app/models/cryptid.js文件中。打开这个文件，在其中添加一些属性：名

称（name）、神秘生物类型（物种，cryptidType）、档案图片（profileImg）和目击记录（sightings）：

```
import DS from 'ember-data';

export default DS.Model.extend({
  name: DS.attr('string'),
  cryptidType: DS.attr('string'),
  profileImg: DS.attr('string'),
  sightings: DS.hasMany('sighting')

});
```

DS（即Data Store）是Ember Data提供的对象，它提供了一个attr方法用来为模型定义属性。数据源传入的数据会通过attr方法返回：如果调用attr时传入了类型作为参数，那么它返回的数据将被强制转化成该类型；如果没传参数，数据则会保持原样。

Ember内建了一些数据类型：字符串（string）、数字（number）、布尔值（boolean）以及日期（date）。此外，也可以通过transforms方法自定义数据类型，这个知识点会在下一章中介绍。

attr的第二个参数是可选的，可以在这个参数中使用defaultValue定义属性的默认值。例如：

```
name: DS.attr('string', {defaultValue: 'Bob'}),
isNew: DS.attr('boolean', {defaultValue: true}),
createdAt: DS.attr('date', {defaultValue: new Date()}),
numOfChildren: DS.attr('number', {defaultValue: 1})
```

这里将神秘生物的名称、类型、图片等属性定义为字符串类型。（为什么图片也是字符串？因为这里存储的是图片的路径，而不是图片本身。）

sightings属性则用了不同的方法来定义数据类型：hasMany。hasMany是Ember Data提供的关系方法之一，用于定义关联关系。当通过RESTful API查询神秘生物时，相关联的目击记录会以id数组的形式返回，每个id代表一条目击记录实例。

Ember Data提供了多种描述关联关系的方法：一对一、一对多、多对多。

关　系	关联模型	被关联模型
一对一	DS.hasOne	DS.belongsTo
一对多	DS.hasMany	DS.belongsTo
多对多	DS.hasMany	DS.hasMany

在定义关系时，第一个参数是待关联的模型名称，第二个参数与attr的第二个参数一样，也是一个可选的配置对象，它包含一个async属性，默认值为true 。在Tracker应用中，每个神秘生物对应多条目击记录（好奇为什么会有这么多人见过它们）。

在定义了关联关系后，当从服务端查询特定的模型数据时也会同时查询关联模型的数据。例如，当查询神秘生物时，会同时查询相关联的目击记录。反之亦然，每条目击记录也会**关联到**一种神秘生物。如果async参数配置为true（默认值），则针对模型数据的查询和针对关联数据的查询会分别发送到各自API接口的请求。

如果API支持一次性发送全部数据，可以将async设置为false。在Tracker应用中就用默认值true。

接下来修改目击者模型app/models/witness.js，为它添加一些属性：

```
import DS from 'ember-data';

export default DS.Model.extend({
  fName: DS.attr('string'),
  lName: DS.attr('string'),
  email: DS.attr('string'),
  sightings: DS.hasMany('sighting')
});
```

在这段代码中定义了目击者对象，它包含了姓（lName）、名（fName）、邮箱（email）和目击记录（sightings，与目击者是多对多的关系）等属性。

最后，修改目击记录模型app/models/sighting.js。对于每条目击记录需要关注的问题有：这是哪种生物，目击的时间、地点、目击者分别是什么，这次目击被记录的日期。在记录中加入这些属性：

```
import DS from 'ember-data';

export default DS.Model.extend({
  location: DS.attr('string'),
  createdAt: DS.attr('date'),
  sightedAt: DS.attr('date'),
  cryptid: DS.belongsTo('cryptid'),
  witnesses: DS.hasMany('witness')
});
```

定义目击记录模型和定义其他模型并无不同，都是使用字符串描述属性。目击记录中的location属性是需要用户手动输入的，而createdAt和sightedAt会根据数据录入数据库的时间在服务端自动添加。

在定义cryptid属性时，用到了新方法DS.belongsTo('cryptid')，这个方法用来描述一对多的关联关系。在应用中，一个神秘生物实例对应多条目击记录——这也很好理解，毕竟一种生物可能被看到多次。

21.2 创建记录

在应用初始化时，Ember Data会创建用于本地存储的store对象。this.store对象包含了对模型记录进行创建、查询、修改、删除等操作的方法。这个store对象会被注入到所有的路由、控制器以及组件中，在路由等对象的相关方法中可以通过this访问store。

创建记录的方法是this.store.createRecord，它需要两个参数：模型名称（字符串）和记录数据（对象）。

打开app.routes/sightings.js文件，删除其中的模拟数据，然后添加三条新的目击记录，这些新

记录中包含了location属性（字符串）和sightedAt属性（通过new Date创建的Date对象）：

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    return [
      {
        id: 1,
        location: 'Asilomar',
        sighted_at: new Date('2016-03-07')
      },
      ...
      {
        id: 6,
        location: 'Asilomar',
        sightedAt: new Date('2016-03-07')
      }
    ];
    let record1 = this.store.createRecord('sighting', {
      location: 'Atlanta',
      sightedAt: new Date('2016-02-09')
    });

    let record2 = this.store.createRecord('sighting', {
      location: 'Calloway',
      sightedAt: new Date('2016-03-14')
    });

    let record3 = this.store.createRecord('sighting', {
      location: 'Asilomar',
      sightedAt: new Date('2016-03-21')
    });

    return [record1, record2, record3];
  }
});
```

在第20章中，目击记录模型同样返回了一个数组。但是现在情况略有不同，数组中不再是简单的JavaScript对象，而是目击记录的实例。执行ember server启动服务器，访问http://localhost:4200/sightings查看新添加的记录（如图21-1所示）。

从这个例子可以看出，创建Ember Data模型与创建普通的JavaScript对象并没有什么不同，但使用Ember Data模型的好处是它提供了许多额外的方法，例如get和set。

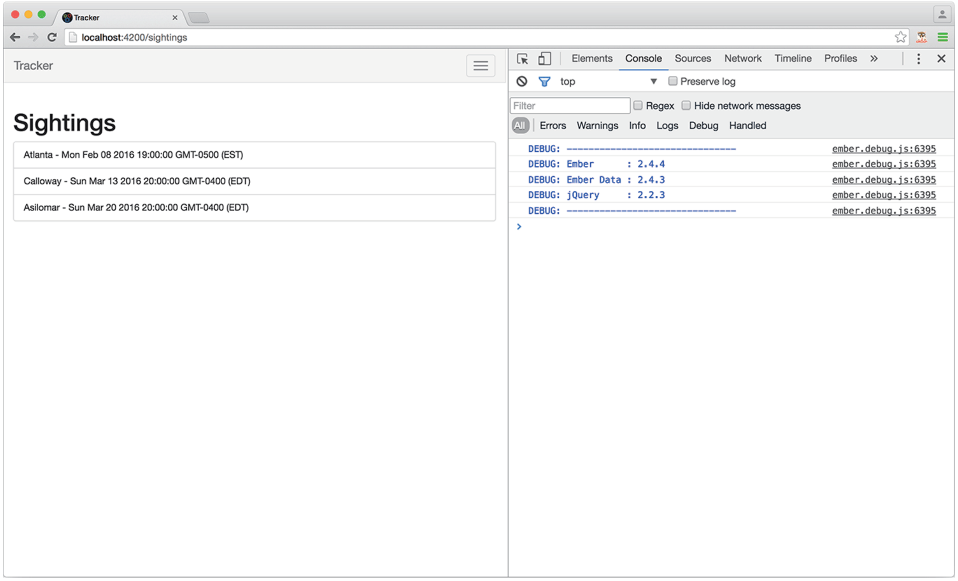


图21-1　创建目击记录

21.3　get 和 set

`Ember.Object`是Ember Data模型记录实例的核心，其中就定义了`get`和`set`方法。不同于大部分语言，JavaScript不会在对对象实例取值或赋值时强制要求使用getter或setter方法。于是Ember实现了一套与getter和setter类似的逻辑，并且强制使用，目的是能够在属性发生改变时执行特定的函数。进一步讲，Ember会在调用`set`方法时触发事件。另外，使用getter可以更加明确读取属性的意图。

`get`方法只接受一个参数，即属性名称，它返回属性的值。用app/routes/sightings.js模型试试。

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    let record1 = this.store.createRecord('sighting', {
      location: 'Atlanta',
      sightedAt: new Date('2016-02-09')
    });
    console.log("Record 1 location: " + record1.get('location') );
    ...
    return [record1, record2, record3];
  }
});
```

将开发者工具切换到Console标签，保持开启不要关闭，然后刷新浏览器。查看控制台中Ember打印的日志信息，最后一条是“Record 1 location: Atlanta”。

继续编辑app/routes/sightings.js，在创建记录和打印日志之间的位置使用set方法修改record1的location属性。

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    let record1 = this.store.createRecord('sighting', {
      location: 'Atlanta',
      sightedAt: new Date('2016-02-09')
    });
    record1.set('location', 'Paris, France');
    console.log("Record 1 location: " + record1.get('location'));
    ...
    return [record1, record2, record3];
  }
});
```

刷新浏览器，在控制台能看到修改之后的值："Record 1 location: Paris, France"（如图21-2所示）。

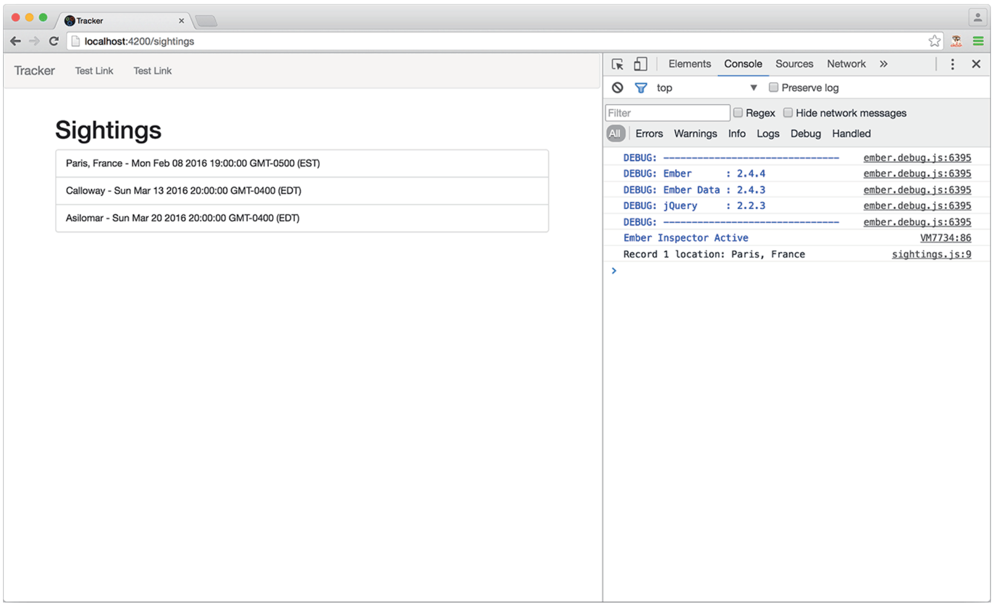


图21-2　通过set设置location属性

这只是get和set最基本的用法。如果要修改的模型记录的属性是通过hasMany或belongsTo定义的，那么可以使用其他模型记录为该记录的属性赋值，这样两个模型之间就会建立关联关系。

21.4　计算属性

模型的计算属性对模板和组件极为重要。可以使用Ember.computed方法定义计算属性，它

会对一些属性的值进行计算并返回结果。比如，调用下面的代码，计算属性会返回first_name属性值的小写形式：

```
Ember.computed('first_name', function(){
  return this.get('first_name').toLowerCase();
});
```

在这个例子中，Ember.computed就像事件监听器一样，时刻监听着first_name属性的变化。但是你既不需要定义监听器或者手动触发事件，也不需要对原有的first_name属性做任何修改，要做的只是添加一个计算属性。

计算属性在Ember中的运用相当广泛，第25章还会为组件创建计算属性。计算属性既可以像上面的例子一样作为**装饰器**用于视图或组件中，也可以用于在模型中检索深层数据。

“装饰”的意思是将数据按某种形式进行格式化，例如上面例子中的大小写转换。API中返回的数据通常都需要经过格式转化才能使用。装饰器本质上是个函数，它将输入的数据按应用（主要是视图）的需求转化成对象或者对象数组的形式。一般来讲，数据格式化的结果不会返给数据库，所以通常在控制器中使用装饰器。除非数据在所有页面中都需要使用，并且在数据库中也没有经过格式化。

编辑app/models/witness.js文件，为目击者模型添加一个fullName计算属性。

```
import Ember from 'ember';
import DS from 'ember-data';

export default DS.Model.extend({
  fName: DS.attr('string'),
  lName: DS.attr('string'),
  email: DS.attr('string'),
  sightings: DS.hasMany('sighting'),
  fullName: Ember.computed('fName', 'lName', function(){
    return this.get('fName') + ' ' + this.get('lName');
  })
});
```

（如果编译服务报错，请检查一下是不是漏掉了sightings属性后面的逗号，这是一个常犯的错误。）

为目击者模型添加的这个属性本身是一个函数，它会在fName或lName发生改变时执行。计算属性可以接受任意多个参数，除了最后一个参数是用来返回计算结果的回调函数外，其余参数都是触发回调函数的属性名。

修改app/routes/witnesses.js文件，添加一条目击者记录，测试刚添加的计算属性能否正常工作：

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    let witnessRecord = this.store.createRecord('witness', {
      fName: "Todd",
      lName: "Gandee",
      email: "fake@bignerdranch.com"
```

21

```
      });
      return [witnessRecord];
    }
  });
```

此外，为了使目击者数据在页面中展示，需要修改app/templates/witnesses.hbs模板，在其中使用{{#each}}进行数据迭代。{{#each}}的用法和在app/templates/sightings/index.hbs中的用法相同。

```
{{outlet}}
<h1>Witnesses</h1>
<div class="row">
  {{#each model as |witness|}}
    <div class="col-xs-12 col-sm-6 col-md-4">
      <div class="well">
        <div class="thumbnail">
          <div class="caption">
            <h3>{{witness.fullName}}</h3>
            <div class="panel panel-danger">
              <div class="panel-heading">Sightings</div>
            </div>
          </div>
        </div>
      </div>
    </div>
  {{/each}}
</div>
```

访问http://localhost:4200/witnesses页面查看效果（如图21-3所示）。

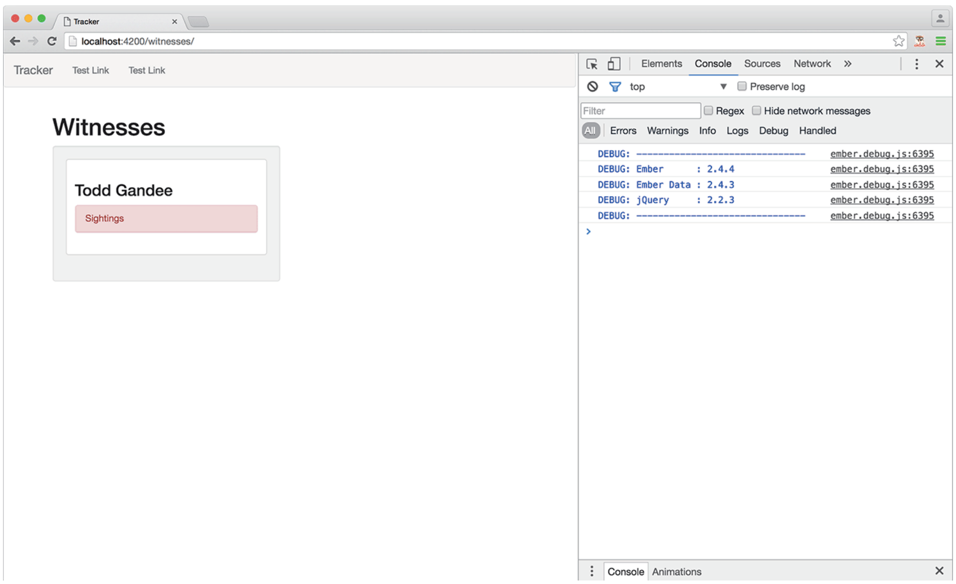


图21-3　目击者列表

页面中展示了目击者列表（虽然目前只有一条记录）和通过计算属性得到的目击者姓名（在添加目击者记录时生成的）。可以尝试在model函数返回之前使用witnessRecord.set修改记录中的姓或名，然后查看修改后的效果。

经过这几章的学习，你已经对Ember有了不少了解——你学会了定义模型、创建记录、创建计算属性、设置和获取属性值等。很快你还会学习如何使用API对记录进行增删改查操作。

下一章将介绍如何对数据使用适配器、序列化器和变换器，用于将数据模型与服务端数据进行关联。

21.5 延展阅读：检索数据

前面提到过，store负责管理和检索数据。在上一章中，我们在目击记录路由（SightingRoute）的model回调中以数组的形式返回数据。而在下一章中，我们将在路由中使用this.store.findAll等检索方法，以Promise的形式返回数据。

下表是Ember提供的数据检索方法，用于从API获取数据，将其存储在内存，并返回给调用者。

请求类型	检索全部数据	检索单条数据
获取远程和本地数据	findAll	findRecord
只获取本地数据	peekAll	peekRecord
获取过滤后的数据	query	queryRecord

Ember提供了多种获取数据的模式：远程和本地、仅本地、经过过滤的远程和本地。其中最常用的是findAll和findRecord，它们接受的参数与API需要的参数类似，因为Ember Data也是通过请求项API接口进行数据查询的。

findAll方法唯一的参数就是模型名称。例如，当需要查询全部目击者时，可以使用findAll('witness')，注意这里的参数是单数形式（即参数是模型名）。但是在进行Ajax请求时，Ember Data会自动以它的复数形式构造URL——/witnesses/。

findRecord比findAll多一个参数，用来指示记录的标识符，通常指id字段。例如当调用this.store.findRecord('witness', 5)时会向/witnesses/5发起数据请求。

peekAll和peekRecord需要的参数和findAll、findRecord相同。不同的是，调用它们会直接返回数据，而不是Promise对象。

从API查询数据也是表单请求的一种形式。如果API接口支持查询参数，那么就可以使用query和queryRecord。和其他方法一样，该方法的第一个参数也是模型名称，第二个参数是一个对象，其中的键值对会在发送请求时转化成查询字符串。调用query会返回所有符合条件的数据。如果你确定结果只有一条数据则可以使用queryRecord。

例如，调用下面这行代码：

```
this.store.query('user',{fName: "todd"})
```

会产生/users/?f_name=todd这样的请求。或者：

```
this.store.queryRecord('user', {email: 'me@test.com'})
```

产生的请求是：/users?email=me@test.com。

store中的这些方法全都使用了适配器，这部分内容在下一章会详细介绍。

21.6　延展阅读：保存或删除数据

在增加和查询之后，接下来要学习的逻辑是修改和删除。（一般记作CRUD，create、read、update、destory，即创建、查询、修改、删除）。[①]模型实例中包含save和destoryRecord两个方法，可用于通过适配器发送请求，从而更新数据存储。它们的返回值是一个Promise对象，所以我们在.then中处理返回的数据。

前面曾经讲过，可以通过set方法对模型记录的属性进行修改。但set只会修改本地的值，所以在修改后本地数据与服务端数据之间会产生差异。为了解决这个问题，在使用set后还需对数据进行保存。保存数据可以通过modelRecord.save实现。在保存时，模型会通过store向API接口发送请求，请求的类型为POST或PUT，根据记录的状态决定。

这里没有提到查询，store在发送查询请求时会使用get方法（请求类型为GET）。而在保存数据时，如果该数据在服务端已经存在（修改数据）就使用PUT，反之（新增数据）则使用POST。

同样，在调用createRecord方法时，数据只添加到内存中，而没有保存到数据库，所以还需要调用save方法来创建和更新服务端数据。[②]

最后一个操作是删除记录。与获取和更新记录的方法相似，modelRecord.destroyRecord使用请求方法（请求类型是DELETE）来删除服务端记录。与save类似，destroyRecord实际上也集成了两步操作：deleteRecord和save。这里的deleteRerord方法只是从内存中删除数据。相比而言，自然是destroyRecord更加实用，它只需要一行代码就可以完成两步操作。

21.7　初级挑战：修改计算属性

目前的姓名属性只简单地对fName和lName进行了拼接。请对它进行一些修改，使用email和fName属性，生成与Todd - tgandee@bignerdranch.com相同格式的字符串进行展示。

21.8　中级挑战：对新的目击记录进行标记

为目击记录模型添加一个布尔属性：isNew，默认值defaultValue为false。在已经创建的目击记录中任选一条，将其isNew属性设置为true。打开Chrome的调试器，查看sightings路由，

① 中文常用“增删改查”，即增加、删除、修改、查询。——译者注

② 这里原文中没有明确指出，事实上save有两种用法：带参数和不带参数。其中带参数的用法集合了set和save的功能，可以查询API文档做进一步了解。——译者注

能看到在所有的数据中只有一条记录的isNew属性值是true。[①]

21.9 高级挑战：添加称呼

目击者需要一个合适的称呼，例如：Gandee先生。请在目击者模型中添加title属性，并想一个有趣并且可以通用的称呼作为默认值。在所有记录中任意选一条不做修改，同时为其他各条记录分别添加title属性。然后添加一个计算属性来展示完整的称呼（姓+称呼）。

维基百科上有一个关于称呼的列表非常不错：en.wikipedia.org/wiki/Title，“Gandee超人”看起来就不赖！

① 事实上这里不能使用isNew作为属性名，因为Ember模型中本身已经包含了isNew属性，再次定义该属性会导致错误。——译者注

第 22 章

数据——适配器、序列化器和变换器

几乎每一款应用都需要与接口进行数据交互，因此将应用连接到数据源也是应用开发中的重要环节。如若不然，哪怕应用拥有再复杂的表格、列表或是事件系统，也都只能是一片空白，因为没有数据可以填充。

这一章将会介绍在Ember中连接数据源的一些基础知识。你将会用到专为这本书开发的API接口，另外还会为应用创建适配器。

与其他几章略有不同，这一章的代码会比较少，而介绍会比较多。本章内容涉及与服务端和数据库的交互，通常它们并不受前端开发人员的控制，这也更接近真实的开发场景。下一章会继续常规的编码之旅。

上一章曾提到过，适配器是应用的译者。在与数据源通信时，应用可能需要以多种方式发送和接受数据。Ember Data内置了JSONAPI适配器和通用的“RESTful” API适配器，它们可以处理大多数常见的情况。

JSONAPIAdapter用于连接数据源，它返回经过格式化的数据。RESTAdapter则提供了一些方法，用来与Rails或Rails的ActiveRecord插件提供的API通信。

创立JSONAPI标准的目的是提供一种可预测并且可扩展的模式用来与服务器交换数据。由于目前有许多服务端语言，每种语言又都有自己惯用的API对象模型，所以JSONAPI被设计成一种可以连接各种语言的桥梁，这样哪怕服务端技术发生变更也不会影响前端应用的正常工作。可以访问JSONAPI官网jsonapi.org了解更多信息。

这一章还会介绍一些安全问题，也就是**序列化器**，它是适配流程中的转化层。最后会介绍**变换器**，它的功能是把数据强制转化成模型所期望的格式。适配器、序列化器和变换器互相协作，它们之间的关系如图22-1所示。

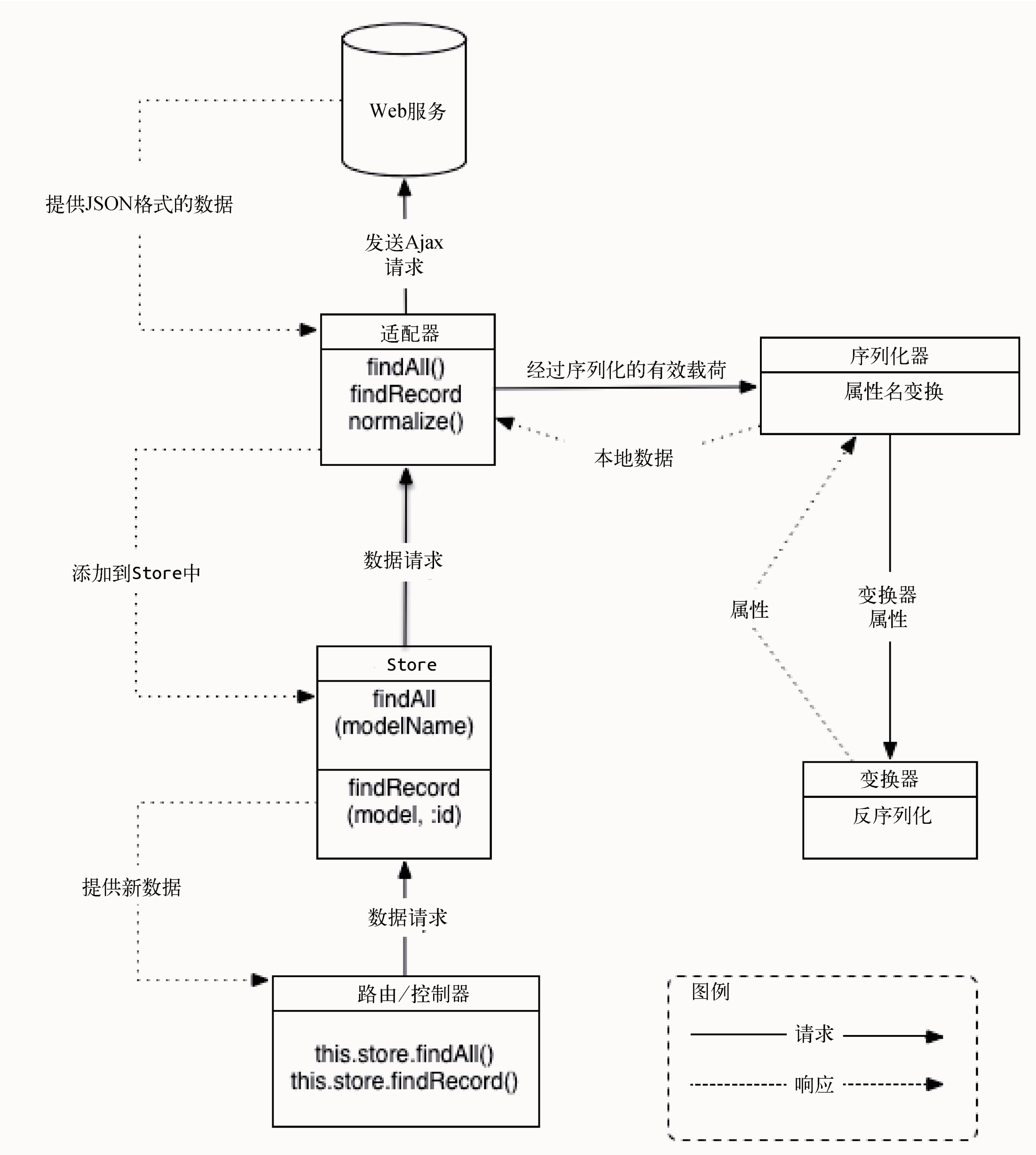


图22-1　适配器、序列化器和变换器

在本章结束时，Tracker的页面应如图22-2所示。

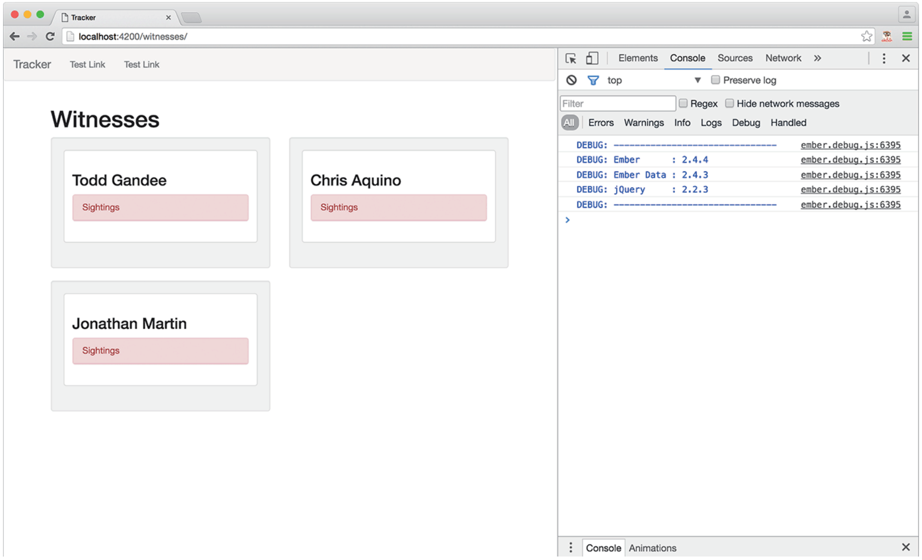


图22-2　本章结束时Tracker的页面截图

22.1　适配器

Ember团队在开发Ember框架时设计了一些特定的模式，适配器占据了其中一大部分。store使用JSONAPIAdapter与REST API进行通信。所有请求都携带模型名称和属性数据，它们会被发送到目标域名下的某个路径。

JSONAPIAdapter需要host和namespace两个属性的值，以便生成Ajax请求的URL。接着适配器会发出Ajax请求，接受JSON响应，但响应数据必须具有特定的结构。举个例子，向目击者列表API发起GET请求，收到的响应结构应该是这样的：

```
{
"links": {
  "self": "http://bnr-tracker-api.herokuapp.com/api/witnesses"
},
"data": [
  {
    "id": "5556013e89ad2a030066f6e0",
    "type": "witnesses",
    "attributes": {
      "lname": "Gandee",
      "fname": "Todd"
    },
    "links": {
      "self": "/api/witnesses/5556013e89ad2a030066f6e0"
    },
    "relationships": {
      "sightings": {
```

```
        "data": [],
        "links": {
          "self":
          "/api/witnesses/5556013e89ad2a030066f6e0/relationships/sightings"
          }
        }
      }
    }
  ]
  }
```

响应中的每条数据都包含一个对象的type属性，它是请求的模型名称，用于解析模型的类型。另外每条数据也都包含id属性，它是数据的主键，用于区分不同的记录。

首先创建一个适配器：

```
ember g adapter application
```

应用要与Big Nerd Ranch Tracker API进行通信。前面提到过，传入JSONAPIAdapter的参数需要包含目标主机的URL（host）和命名空间（namespace），命名空间会在发送请求时附加到URL的最后。打开app/adapters/application.js，修改其中的host和namespace属性。

```
import DS from 'ember-data';

export default DS.JSONAPIAdapter.extend({
  host: 'https://bnr-tracker-api.herokuapp.com',
  namespace: 'api'
});
```

与Ember中的其他类一样，适配器也具有一定的命名规则，同时它也可以根据模型API的需求进行定制。这样做的好处是，可以使用一个定制的适配器处理非标准的数据，而不需要让全部模型向边缘案例妥协。

写在app/adapters/application.js中的配置是全局配置，它对所有的数据请求都有效。在Tracker应用中使用这个配置已经足够了，因为所有API使用的都是相同的主机名和命名空间。但假如这里的目击者模型使用的API主机名或命名空间与其他API不同，那么就需要创建一个app/adapters/witness.js文件，并在其中添加供目击者API使用的专用配置。

接下来需要通过store向API接口发送数据请求。目击者和神秘生物的API已经提前准备好了。

打开app/routes/witnesses.js，将其中的模拟数据替换成上一章讲的数据检索方法。

```
...
  model(){
    let witnessRecord = this.store.createRecord('witness', {
      fname: "Todd",
      lname: "Gandee",
      email: "fake@bignerdranch.com"
    });
    return [witnessRecord];
    return this.store.findAll('witness');
    }
  });
```

在终端中执行`ember server`命令重启应用，然后在浏览器中访问http://localhost :4200/witnesses（如图22-3所示）。

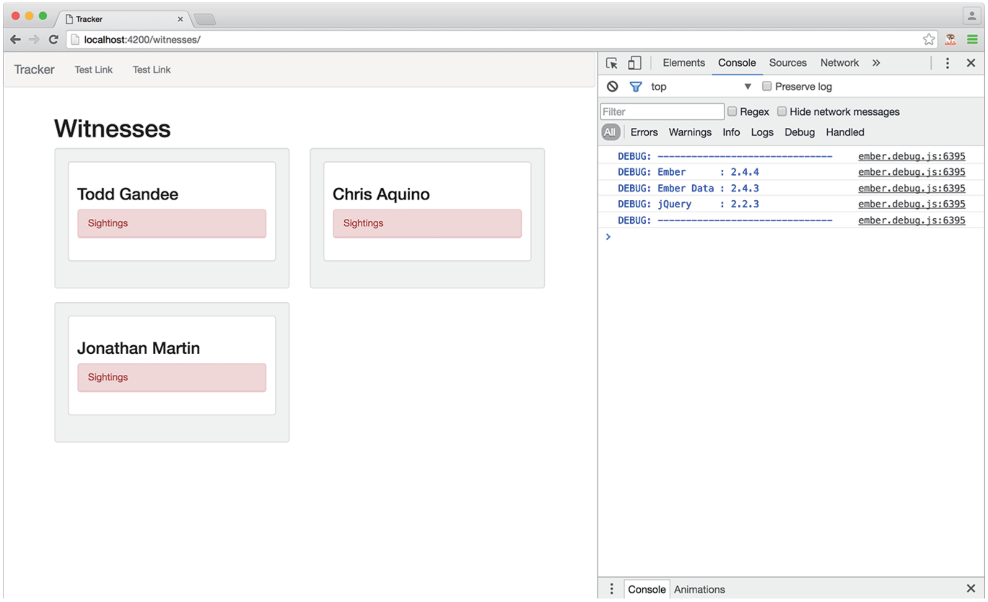


图22-3　目击者列表

跟Ember的大多数生命周期流程一样，适配器中也包含了许多方法，用于从API获取数据然后发送到`store`以及路由、控制器和模板。定义特殊适配器（如JSONAPIAdapter）的目的是提供一种普遍适用的模式，只需要模型的输入/输出数据的格式符合要求即可使用。Tracker应用的服务端使用了Node.js构建的服务器，数据库使用MongoDB，另外通过`json-api`模块提供符合JSONAPI标准的接口。

一个稳定的接口（很少出现不符合格式要求的数据）能为开发者带来愉悦的开发体验。当然开发过程中也需要考虑请求中的额外数据，例如认证信息或请求首部等。可以通过定制适配器来处理这些情形。

在早些时候，Ember内置了处理其他数据源（例如localStorage、固定数据等）的适配器。但现在这些适配器不再直接内置，而是以插件的形式提供。如果应用的模型需要使用这些数据源，可以通过Ember CLI安装对应的插件。

在设计适配器时，可以从文档中获取有帮助的信息。重点关注这些方法或属性：`ajaxOptions`、`ajaxError`、`handleResponse`和`headers`。

现在的Tracker应用已经能够向服务器发送查询目击者的请求并接受返回的数据。在继续学习新内容之前，将神秘生物模型也改为使用`this.store.findAll`获取数据。由于目前还没有为神秘生物创建模板，所以暂时需要通过Ember Inspector查看数据。

修改app/routes/cryptids.js：

```
import Ember from 'ember';

export default Ember.Route.extend({
  model(){
    return this.store.findAll('cryptid');
  }
});
```

现在应用能够获取数据了，打开http://localhost:4200/cryptids，在Ember Inspector中查看返回的数据：打开开发者工具，选择Ember标签，点击Data，接着点击cryptid(4)（如图22-4所示）。

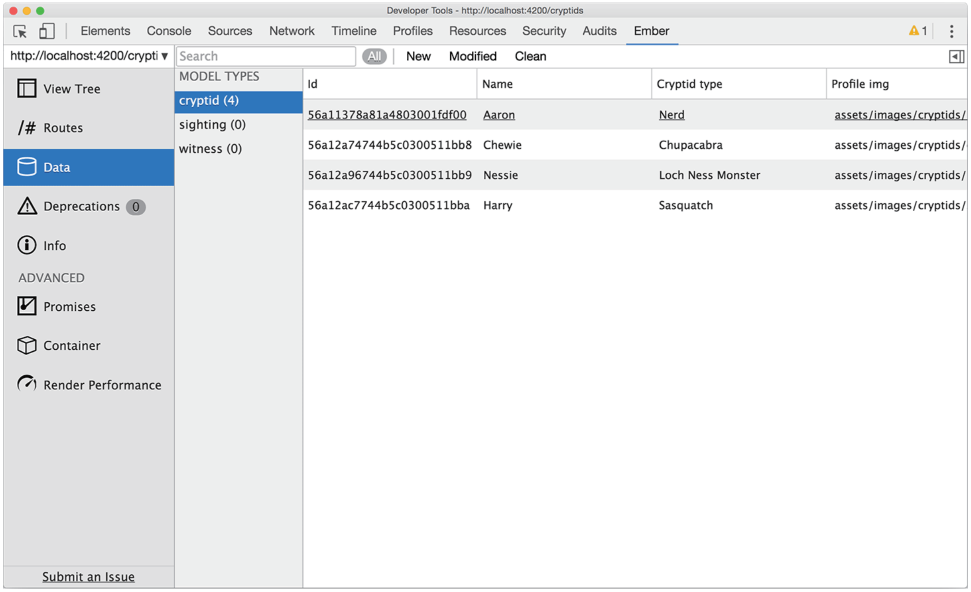


图22-4　神秘生物数据

目前只用到了Ember Inspector查看内存中数据的功能，其他更多的功能还有待探索，例如Ember可以从API请求数据并存储到对应的`store`中。当需要对模型数据进行调试时，跟踪请求路径的过程通常乏味而冗长，这时应该优先使用Ember Inspector。

22.2　内容安全策略

为了能在请求到达服务端之前检测到跨站攻击，Ember在JavaScript中添加了一个安全层，使用的工作标准是内容安全策略（Content Security Policy）。Ember CLI内置了`contentSecurityPolicy`对象用于提供相应的信息。Ember默认的安全策略非常严格，这些策略适用于向应用域名之外的域名（跨域）发起的请求，包括对数据、脚本、图片、样式以及其他类型文件的请求。

Ember提供了一款插件ember-cli-content-security-policy，用于修改默认安全策略。虽然在

Tracker中还用不到它，但最好能对它有所了解。使用这个插件为安全策略设置环境变量十分简单，它提供的安全策略对象与浏览器的内容安全策略规范兼容。一些较新的浏览器会默认使用内容安全策略，以便阻止由于运行恶意代码而引发的跨站脚本攻击和代码注入攻击。

这儿有一个contentSecurityPolicy对象的示例：

```
module.exports = function(environment) {
  ...
  // config/environment.js
  ENV.contentSecurityPolicy = {
    'default-src': "",
    'script-src': "",
    'font-src': "",
    'connect-src': "",
    'img-src': "",
    'style-src': "",
    'media-src': null
  }
  ...
}
```

其中的每行代码都为一种特定的请求类型定义了一份**白名单**，也就是一个包含多条安全URL的集合。其中default-src对所有未明确指定策略的请求生效，它的默认值为null，目的是强制开发者指定白名单。其他的设置（例如script-src和connect-src）对向外部域名（例如https://bnr-tracker-api.herokuapp.com）发送的请求有效。

可以从MDN的Content Security Policy页面或者Ember CLI插件的GitHub仓库获取更多信息。

22.3　序列化器

数据在传入和传出时，JSON结构会经历序列化和反序列化。适配器在数据流入或流出store时使用序列化器来构建请求数据或发送响应数据。

创建序列化器的命令是ember g serialize [应用或模型名称]，执行这个命令会创建序列化器文件。在其中添加样例代码：

```
import DS from 'ember-data';

export default DS.JSONAPISerializer.extend({
});
```

序列化器本身是一个对象，会作为serializer属性被添加到适配器对象上。如果在应用中没有定义序列化器，则Ember会使用默认的适配器和序列化器，也就是JSONAPIAdapter和JSONAPIS erializer。

在应用引入新的序列化器时，这个序列化器会被对应的app/adapters/application.js文件用作defaultSerializer。以模型的名字作为参数执行命令ember g serializer可以为该模型定制数据的序列化器。

当使用JSONAPIAdapter时，只有在API不符合JSONAPI规范或者应用中包含边缘案例时，才

需要修改默认配置。如果需要在项目中修改请求或响应的数据，可以查阅Ember Data文档中的这些方法：keyForAttribute、keyForRelationship、modelNameFormPayloadKey和serialize。

keyForAttribute的功能是将模型的属性名转化成请求数据需要的键名。它需要3个参数：key、typeClass和method。对JSONAPISerializer使用这个方法后会返回**中划线分割**的键名，换句话说，这个方法会将键名里下划线分割的样式和驼峰样式都转化成中划线分割的样式。例如，模型中包含一个属性first_name，那么它在请求对象中会被转化为first-name。反过来，如果API要求的格式是first_name，那么需要修改keyForAttribute方法来解决这个命名问题。

keyForRelationship也是相似的逻辑，只不过它针对的是描述关系的键名。如果使用belongsTo或hasMany定义模型间的关系，并且模型名中包含下划线，那么需要修改这个方法以便于JSONAPISerializer使用。有些API要求描述关系的字段以_id或_ids结尾，也可以通过这个方法来实现。

Tracker使用的接口bnr-tracker-api对键名的要求与JSONAPI一致，也就是中划线分割，如cryptid-type，所以Tracker应用并不需要使用序列化器。这里提供的例子只是用来展示序列化器的用法：使用Ember提供的工具方法Ember.String.underscore改变请求和响应数据的属性名。

```
import Ember from 'ember';
import DS from 'ember-data';
var underscore = Ember.String.underscore;
export default DS.JSONAPISerializer.extend({
  keyForAttribute(attr) {
    return underscore(attr);
  },
  keyForRelationship(rawKey) {
    return underscore(rawKey);
  }
});
```

Ember在Ember.String对象中提供了许多操作字符串的方法。Ember.String.underscore的功能是将使用空格分割或驼峰形式的字符串转换为全部小写且使用下划线分割的形式。

序列化器会在数据请求的流程中调用，例如在this.store.findAll('witness')中。如果需要查看数据请求时的回调流程，也可以在序列化器中加入调试代码，这样就可以看到流入和流出序列化器的数据。

来看一个示例：

```
import Ember from 'ember';
import DS from 'ember-data';

var underscore = Ember.String.underscore;

export default DS.JSONAPISerializer.extend({
  keyForAttribute(attr) {
    let returnValue = underscore(rawKey);
    debugger;
    return returnValue;
    return underscore(rawKey);
```

```
    },
    keyForRelationship(rawKey) {
      return underscore(rawKey);
    }
  });
```

在应用接入新的API时，这种调试方案非常有用。当需要调整attr的参数以便对键名进行序列化时，用这种调试方案可以访问Ember.String对象。感谢Ember和Ember社区为各种API模式开发了适配器和序列化器。

22.4　变换器

Ember Data提供了转化数据的能力，可以将API提供的数据转化成应用需要的格式。第21章曾使用过Ember内置的变换器DS.attr('string')、DS.attr('boolean')、DS.attr ('number')和DS.attr('data')。

也可以为应用添加自定义的变换器，添加之后通过DS.attr使用新定义的属性类型。变化器接受一个值并将它转化成指定的类型后返回，有些类似于JavaScript中的类型强制转化。

这有一个使用DS.attr('object')变换器的例子：

```
export default DS.Transform.extend({
  deserialize(value) {
    if (!Ember.$.isPlainObject(value)) {
      return {};
    } else {
      return value;
    }
  },
  serialize(value) {
    if (!Ember.$.isPlainObject(value)) {
      return {};
    } else {
      return value;
    }
  }
});
```

需要为变换器定义两个方法：deserialize和serialize。deserialize的功能是判断传入应用的数据是否是对象类型，如果是对象则直接返回，反之返回空对象。serialize的功能是判断从应用传出的数据是否是对象类型，如果是对象则直接返回，反之返回空对象。使用变换器的目的是保证发送至API的数据和从API返回的数据都与在模型中定义的数据类型相符。

22.5　延展阅读：Ember CLI Mirage

API不可用是前端工程师在开发过程中经常遇到的问题，如API还未开发、API在开发中、API在开发环境无法访问等。

对于API不可用的情况有一种相对简单的解决方案，即使用静态的文字或者固定的数据替代

请求。但这种方案也有它的弊端，首先是需要修改程序逻辑来适应这些无法访问的数据，其次需要在`this.store`中修改请求，有时还需要修改适配器或序列化器。

Ember CLI提供了一个插件Mirage，它可以将请求代理到特定的API路由上。可以通过创建模型来建立关联关系，创建工厂来生成数据，为指定的响应定义固定数据，也可通过定义CRUD路由拦截发给API的特定请求。当配置工作完成后，这些API便可以模拟出正常工作时的效果。

当Mirage开启时，所有的请求会根据Mirage的配置转发到本地。

在本书出版时，ember-cli-mirage的最新版本是0.2.0-beta.8，短时间内使用Mirage还只能作为一种探索和尝试。可以访问www.ember-cli-mirage.com查看更多信息。

下一章中将汇总前面3章介绍过的内容，完成一款能够接受路由请求并展示数据的应用。第24章将会对数据进行增加、编辑和删除。接下来就能看到Ember Data、适配器和序列化器的威力：你可以在模型上调用`save`和`destroyRecord`，并且整个将数据发送到API的过程都无须你过多费心。

22.6　中级挑战：内容安全

在任何时候都应该为应用添加一个安全层。前面提到过，浏览器为内容安全策略提供了新的API，并且Ember也提供了一个环境对象用于配置安全策略的白名单。请安装这个插件，并根据控制台的错误提示，将应用中用到的所有外部请求都加入白名单。

22.7　高级挑战：Mirage

Ember CLI Mirage对开发者来说是一件称手的兵器。有了它，在开发应用与API的交互时就不用受后端团队API开发进度的限制。

在终端中安装ember-cli-mirage：

```
ember install ember-cli-mirage
```

接下来，在config/environment.js中添加一个环境变量以控制Mirage的开关状态：

```
if (environment === 'development') {
  // ENV.APP.LOG_RESOLVER = true;
  // ENV.APP.LOG_ACTIVE_GENERATION = true;
  // ENV.APP.LOG_TRANSITIONS = true;
  // ENV.APP.LOG_TRANSITIONS_INTERNAL = true;
  // ENV.APP.LOG_VIEW_LOOKUPS = true;
  ENV['ember-cli-mirage'] = {
    enabled: true
  }
}
```

最后，以工厂的形式为目击者和神秘生物接口创建假数据。

可以从例子资源Tracker/Data_Chapter/mirage-example项目的app/mirage目录中查看配置示例，并将其运用到Tracker应用中。我们会经常更新这个示例，确保它能与最新版本的插件兼容。

第 23 章 视图与模板

MVC中的V代表的是**视图**（view）。对Tracker应用来说，视图指的是模板。通过JavaScript的处理，模板可以转化成HTML元素。而通过使用模板，可以在不依赖网络请求的情况下改变页面的DOM结构。

在这一章中，我们会创建一些模板文件并在路由的`model`方法中加入数据检索的逻辑。Ember提供了内置的模板语言和许多辅助函数，这使得使用模板开发页面的工作量远远小于使用常规HTML语法进行开发的工作量。

到本章结束时，Tracker的目击记录列表将如图23-1所示。

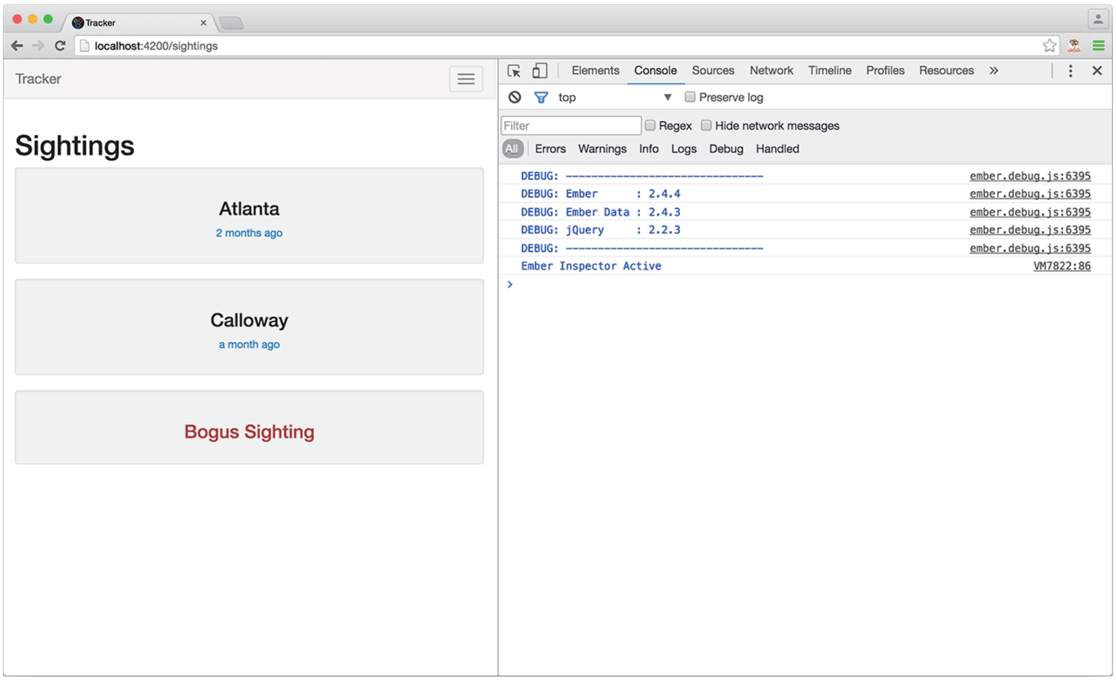


图23-1　目击记录列表

23.1 Handlebars

Handlebars是一种强大的语言，可用于创建动态模板，有些类似于PHP、JSP、ASP和ERB等服务器端使用的模板语言。模板中会包含HTML标签和处理数据对象所使用的分隔符。

Handlebars选用的分隔符是双大括号`{{}}`。可以将数据对象或表达式置于大括号中进而将它们渲染成字符串，也可以借助辅助方法执行简单的逻辑。我们曾见过`{{outlet}}`和`{{#each}}`这两个辅助方法。

Ember最近更新了Handlebars的解释器，取名为HTMLBars。本章的部分内容基于HTMLBars，但同样适用于旧版本的Ember（最低版本为1.13.*x*）。

23.2 模型

在Ember中，模板是由模型数据支撑的。换句话说，数据对象（或对象数组）会作为参数传递给模板，接着模板被渲染成HTML字符串并插入到DOM中。

每个模型对象可以包含多种类型的属性，如字符串、数组或其他对象。在模板里可以通过双大括号来访问模型对象及其属性。

如果需要在模板中显示一个模型属性，例如`name`，那么可以使用`{{model.name}}`。这种语法看起来可能很眼熟，但千万不要被它的外表所迷惑而试图直接在大括号中执行JavaScript语句。

23.3 辅助方法

Handlebars模板本质上只是一个大字符串，模板引擎通过JavaScript函数对它进行解析和替换等操作。当解析器遇到大括号时会解析大括号中的内容，根据其内容返回相应的对象属性或运行函数并返回运行结果。这里的函数就是**辅助方法**。Handlebars内置了一些辅助方法，Ember又根据框架的需要补充了一些。

辅助方法有两种写法。一种是行级写法，形式是`{{[helper name] [arguments]}}`。这里的参数（`arguments`）可以包含多个键值对，例如：

```
{{input type="text" value=firstName disabled=entryNotAllowed size="50"}}
```

另一种是复杂一些的块级写法：

```
{{#[helper name] [arguments]}}
  [block content]
{{/[helper name]}}
```

例如，在模板中添加一个登录链接并且只向未登录用户展示：

```
{{#if notSignedIn}}
  <a href="/">Sign In</a>
{{/if}}
```

可以在块级辅助方法的标签之间传入内容，这些内容会被辅助方法动态渲染出来。这就需要

使用下一节所讲的Handlebars内置的条件语句，它也是一个块级辅助方法。

后面在渲染目击者、神秘生物和导航条（NavBar）等模板时都会用到辅助方法。

23.3.1　条件语句

条件语句使Handlebars模板有了简单的流程控制能力，它们的语法如下：

```
{{#if argument}}
  [render block content]
{{else}}
  [render other content]
{{/if}}
```

也可以使用：

```
{{#unless argument}}
   [render block content]
{{/unless}}
```

条件语句需要一个参数，这个参数会被转化成`true`或者`false`。（`false`、`0`、`""`、`null`、`undefined`和`NaN`会被转化为`false`，其他值被转化为`true`。）

现在回到项目代码中。打开app/templates/sightings/index.hbs文件，在其中添加一条条件语句：当目击记录中存在地点数据时显示该地点，否则显示一条提示信息。

```
<div class="panel panel-default">
  <ul class="list-group">
    {{#each model as |sighting|}}
      <li class="list-group-item">
        {{sighting.location}} - {{sighting.sightedAt}}
      </li>
    {{/each}}
  </ul>
</div>
<div class="row">
  {{#each model as |sighting|}}
    <div class="col-xs-12 col-sm-3 text-center">
      <div class="media well">
        <div class="caption">
          {{#if sighting.location}}
            <h3>{{sighting.location}} - {{sighting.sightedAt}}</h3>
          {{else}}
            <h3 class="text-danger">Bogus Sighting</h3>
          {{/if}}
        </div>
      </div>
    </div>
  {{/each}}
</div>
```

在修改目击记录模板的DOM结构的同时，你还引入了Bootstrap提供的样式，以便能够以更友好的样式输出信息。通过`{{#if}}`和`{{else}}`，实现根据目击记录中是否存在地点信息，从而

渲染出不同的HTML。

现在尝试修改数据，以便测试条件语句的效果。打开目击记录的路由模型app/routes/sightings.js，任选一条记录并将其地点信息修改为空值。

```
...
  model(){
    ...
    let record3 = this.store.createRecord('sighting', {
      location: 'Asilomar',
      sightedAt: new Date('2016-03-21')
    });

    return [record1, record2, record3];
  }
});
```

启动服务器，访问http://localhost:4200/sightings查看修改后的列表（如图23-2所示）。

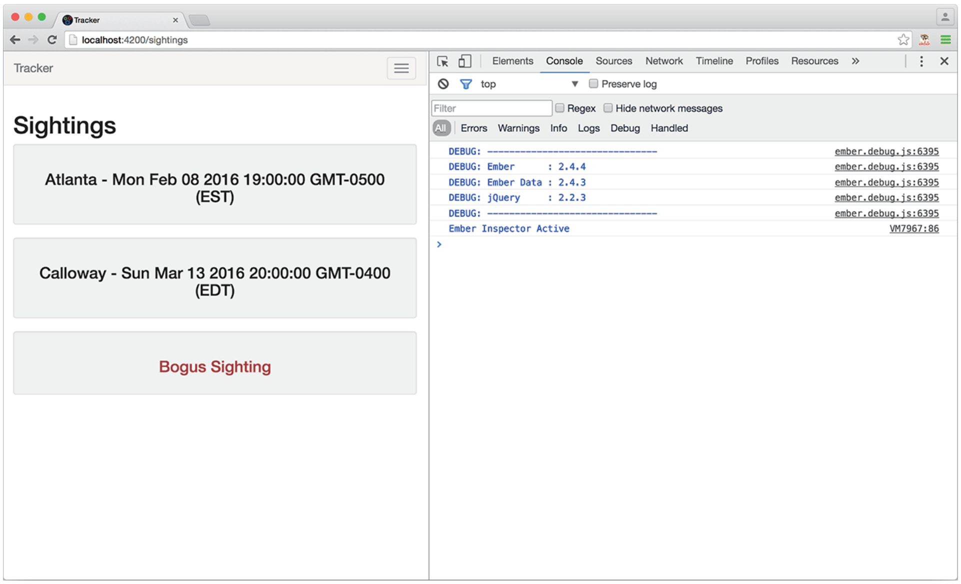


图23-2　伪造的记录

最后一条目击记录的地点信息值是空字符串，所以条件语句将它转化为false，继而渲染出一条警告信息。

23.3.2　{{#each}}循环

首页的模板中使用了{{#each}}辅助方法，它的功能是遍历数组，在每次循环时从数组中取出一个对象实例供循环体使用。{{#each}}参数的格式是array as |instance|，其中array是一个对象数组，而instance是在循环体中访问数组元素时的实例名。{{#each}}只有在array数

组至少包含一个元素时才会对循环体进行渲染。

与`{{#if}}`类似，也可以在`{{#each}}`后使用`{{else}}`，当`array`数组为空时会渲染`{{else}}`后的内容。

修改app/templates/cryptids.hbs模板，使用`{{#each}} {{else}} {{/each}}`渲染神秘生物列表，当列表为空时显示提示信息“No Creatures”。

```
{{outlet}}
<div class="row">
  {{#each model as |cryptid|}}
    <div class="col-xs-12 col-sm-3 text-center">
      <div class="media well">
        <div class="caption">
          <h3>{{cryptid.name}}</h3>
        </div>
      </div>
    </div>
  {{else}}
    <div class="jumbotron">
      <h1>No Creatures</h1>
    </div>
  {{/each}}
</div>
```

和目击记录页面一样，这里也为神秘生物页面加入更友好的样式。通过使用`{{else}}`标签对数组为空的情况进行处理：当神秘生物列表为空时，在页面中渲染一个提示信息。

访问http://localhost:4200/cryptids查看刚创建的神秘生物列表（如图23-3所示）。

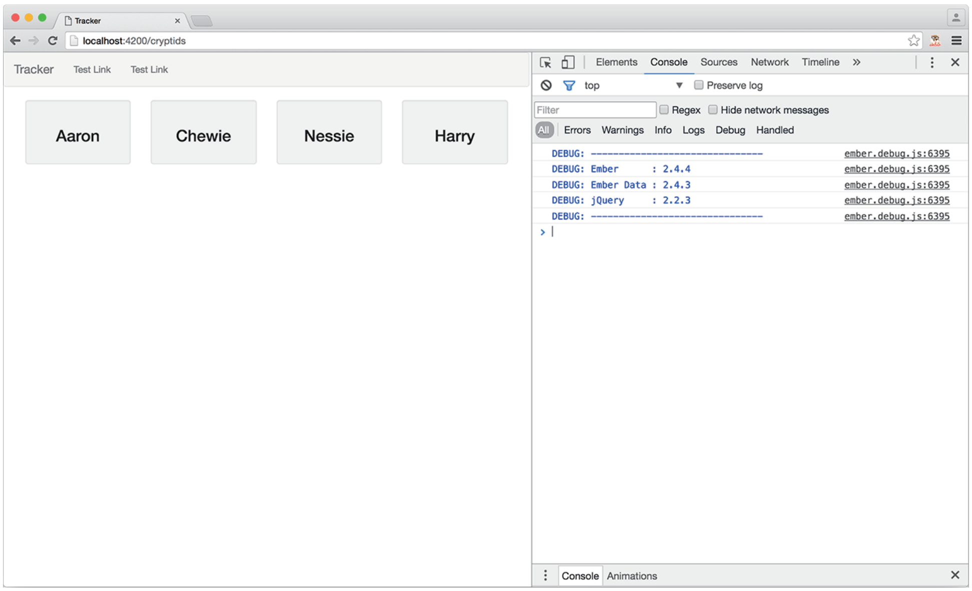


图23-3　神秘生物列表

现在打开app/routes/cryptids.js，把其中返回数据的那一行代码注释掉，模拟一个数据为空的场景。

```
...
  model(){
    // return this.store.findAll('cryptid');
  }
});
```

刷新http://localhost:4200/cryptids，再次查看神秘生物列表（如图23-4所示）。

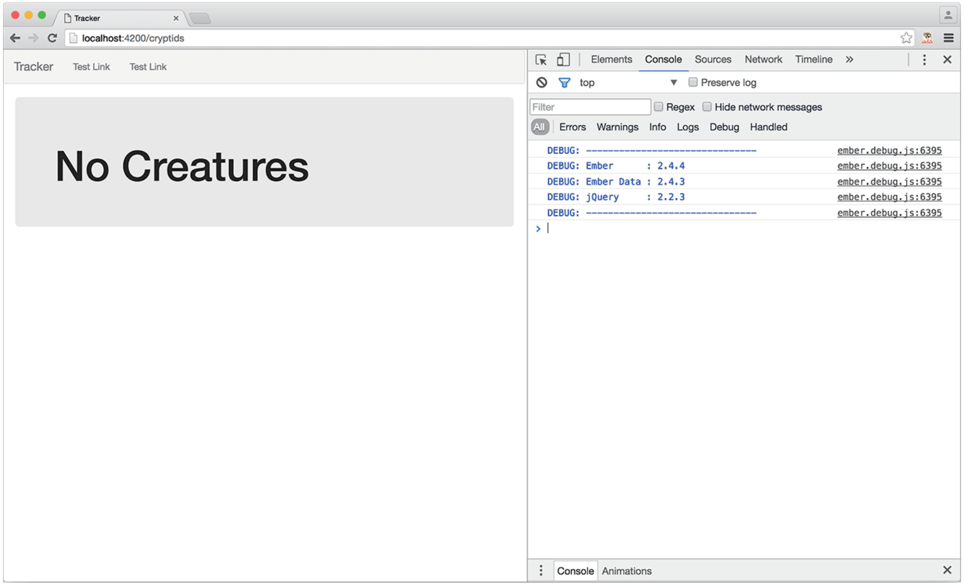


图23-4 神秘生物列表为空

借助`{{each}} {{else}} {{/each}}`，无须使用复杂的逻辑就可以根据数据状态控制页面内容的展示。现在已经验证了条件遍历语句的效果，还原app/routes/cryptids.js文件。

```
...
  model(){
    // return this.store.findAll('cryptid');
  }
});
```

23.3.3 元素属性赋值

使用模板不仅可以在标签之间渲染内容，也同样可以为标签的属性赋值。对于早前版本的Ember，需要使用`{{bind-attr}}`辅助方法才能为属性赋值。不过新版本中引入了HTMLBars，它支持直接使用`{{}}`为元素的属性赋值。

属性赋值常用于元素属性，比如`class`和`src`。神秘生物记录中包含图片路径，所以可以动

态地将模型的属性赋值给图片元素的`src`属性。

修改app/templates/cryptids.hbs，为列表中的神秘生物记录添加一张图片：

```
<div class="row">
  {{#each model as |cryptid|}}
    <div class="col-xs-12 col-sm-3 text-center">
      <div class="media well">
        <img class="media-object thumbnail" src="{{cryptid.profileImg}}"
          alt="{{cryptid.name}}" width="100%" height="100%">
        <div class="caption">
          <h3>{{cryptid.name}}</h3>
        </div>
      </div>
    </div>
  {{else}}
    <div class="jumbotron">
      <h1>No Creatures</h1>
    </div>
  {{/each}}
</div>
```

这里需要把神秘生物的图片素材复制到tracker/public/assets/image/cryptids目录中。Ember服务器在运行时以public目录作为资源的根目录。假如应用需要在生产环境工作，则还需要对资源路径进行进一步配置。但对开发过程来说，可以简单地使用public/assets目录。这些文件会在编译时被复制到dist目录下。

在把应用部署到生产环境时，需要对引用图片的路径进行处理，使用图片的真实路径。在这个DEMO项目中，我们只是简单地把图片保存到同一目录下，并把图片的相对路径记录到数据库中。

在引入HTMLBars之前，如果需要根据一个布尔值决定是否渲染变量，通常会在`{{bind-attr}}`里使用三元操作符。但是现在可以直接使用`{{if}}`。在某些场景中，如根据变量的值渲染页面样式时，这种用法会非常常见。

修改app/templates/cryptids.hbs，使用三元操作处理图片不存在的情况。

```
<div class="row">
  {{#each model as |cryptid|}}
    <div class="col-xs-12 col-sm-3 text-center">
      <div class="media well">
        <img class="media-object thumbnail" src="{{cryptid.profileImg}}"
          src="{{if cryptid.profileImg cryptid.profileImg
          'assets/images/cryptids/blank_th.png'}}"
          alt="{{cryptid.name}}" width="100%" height="100%">
        <div class="caption">
        ...
```

与块级的`{{#if}}`不同，行级的`{{if}}`不会输出内容块。它会根据第一个参数的真假选择输出第二个或者第三个参数。

对这段代码来说，首先要计算第一个参数`{{cryptid.profileImg}}`的真假。如果为真，输

出神秘生物的图片路径，反之则输出一个占位图。

行内的辅助方法支持使用任意变量作为参数，甚至也支持JavaScript的基本类型，如字符串、数组、布尔值等。

在查看页面效果之前，首先对app/routes/cryptids.js做一些修改，在beforeModel钩子中添加一条不包含图片的神秘生物记录。

```
import Ember from 'ember';

export default Ember.Route.extend({
  beforeModel(){
    this.store.createRecord('cryptid', {
      "name": "Charlie",
      "cryptidType": "unicorn"
    });
  },
  model(){
    return this.store.findAll('cryptid');
  }
});
```

现在刷新http://localhost:4200/cryptids页面，看看添加图片之后的列表（如图23-5所示）。

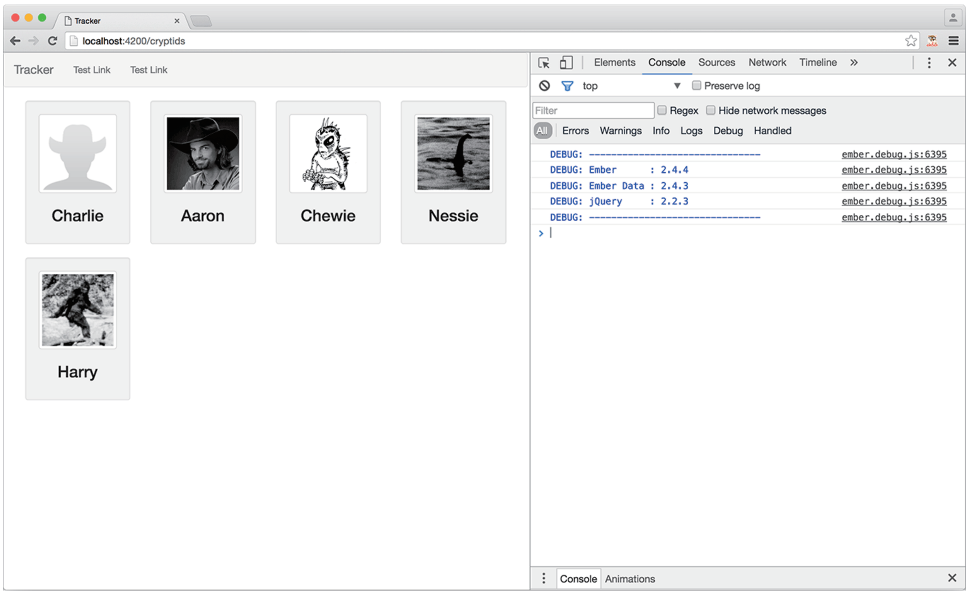


图23-5　带有图片的神秘生物列表

23.3.4　链接

在第20章就讨论过，对于一款基于浏览器的应用，路由是其独特的标志。对具体的应用来说，

Ember通过监听事件钩子对路由进行管理。因此，需要使用{{#link-to}}来创建链接，它的参数是目的路由的名称，调用的结果是生成一个<a>元素。例如调用{{#link-to 'index'}} Home{{/link-to}}会创建一个到主页的链接。

尝试使用{{#link-to}}替换掉导航中的链接。

打开app/templates/application.hbs文件，将其中的假链接替换成指向目击记录、神秘生物和目击者的链接：

```
...
      <div class="collapse navbar-collapse" id="top-navbar-collapse">
        <ul class="nav navbar-nav">
          <li>
            <a href="#">Test Link</a>
          </li>
          <li>
            <a href="#">Test Link</a>
          </li>
          <li>
            {{#link-to 'sightings'}}Sightings{{/link-to}}
          </li>
          <li>
            {{#link-to 'cryptids'}}Cryptids{{/link-to}}
          </li>
          <li>
            {{#link-to 'witnesses'}}Witnesses{{/link-to}}
          </li>
        </ul>
      </div><!-- /.navbar-collapse -->
```

现在这些导航已经可以链接到各个页面了。点点链接，再点点返回键。应用可以正常工作了，真棒！

接下来为神秘生物列表中的照片也加上链接，链接指向神秘生物的详情页。在这个例子中，{{#link-to}}接受多个参数，通过这些参数对链接进行定制。打开app/templates/cryptids.hbs文件，在<img>标签外包裹上指向神秘生物详情页的链接，并且传入cryptid.id作为链接的第二个参数。

```
...
      <div class="media well">
        {{#link-to 'cryptid' cryptid.id}}
          <img class="media-object thumbnail"
          src="{{if cryptid.profileImg cryptid.profileImg
          'assets/images/cryptids/blank_th.png'}}"
          alt="{{cryptid.name}}" width="100%" height="100%">
        {{/link-to}}
        <div class="caption">
          <h3>{{cryptid.name}}</h3>
        </div>
      </div>
...
```

添加的这些链接格式是`cryptids/[cryptid_id]`，指向神秘生物详情页。现在需要对router.js进行修改，使神秘生物详情页路由（CryptidRoute）能接受并处理动态参数。

```
...
Router.map(function() {
  this.route('sightings', function() {
    this.route('new');
  });
  this.route('sighting', function() {
    this.route('edit');
  });
  this.route('cryptids');
  this.route('cryptid', {path: 'cryptids/:cryptid_id'});
  this.route('witnesses');
  this.route('witness');
});
...
```

尝试点一下神秘生物列表中的图片，如果能看到一个空白页面，就说明刚添加的路由已经能够正常工作了。这个链接会跳转到神秘生物页面，这个页面使用的是app/templates/cryptid.hbs模板。由于这个模板目前还是空的，所以页面也是一片空白。

但如果点击的是Charlie（一头独角兽）的图片，应该会看到一条错误提示。回想一下，这条记录是在`beforeModel`钩子中添加的，记录中没有`id`属性，也就意味着传入`{{#link-to}}`的值为`null`。

现在将`beforeModel`钩子移除，因为正常情况下所有的神秘生物都应该通过添加神秘生物的表单页面（会在下一章中创建这个页面）进行添加。

打开app/routes/cryptids.js，删掉`beforeModel`：

```
...
  beforeModel(){
    this.store.createRecord('cryptid', {
      "name": "Charlie",
      "cryptidType": "unicorn"
    });
  },
  model(){
    return this.store.findAll('cryptid');
  }
...
```

接下来修改app/routes/cryptid.js，在其中添加对服务端的数据请求：

```
import Ember from 'ember';

export default Ember.Route.extend({
  model(params){
    return this.store.findRecord('cryptid', params.cryptid_id);
  }
});
```

23

路由接受`cryptid_id`参数后又把它传给模型，作为`findRecord`的参数发起查询请求。

打开app/templates/cryptid.hbs，这个模板用来渲染神秘生物的详情页，其中展示了神秘生物的图片和名称。

```
{{outlet}}
<div class="container text-center">
  <img class="img-rounded" src="{{model.profileImg}}" alt="{{model.type}}">
  <h3>{{model.name}}</h3>
</div>
```

这一次传递给模板的参数不再是数组，而是一个对象。它是一个神秘生物的实例，通过调用`this.store.findRecord`方法返回。在模板中使用`model`访问这个对象，通过`{{model.[property-name]}}`可以获取对象的属性。

打开浏览器，通过顶部导航打开神秘生物页面，在列表中点击任意一个神秘生物的图片查看详情页（如图23-6所示）。

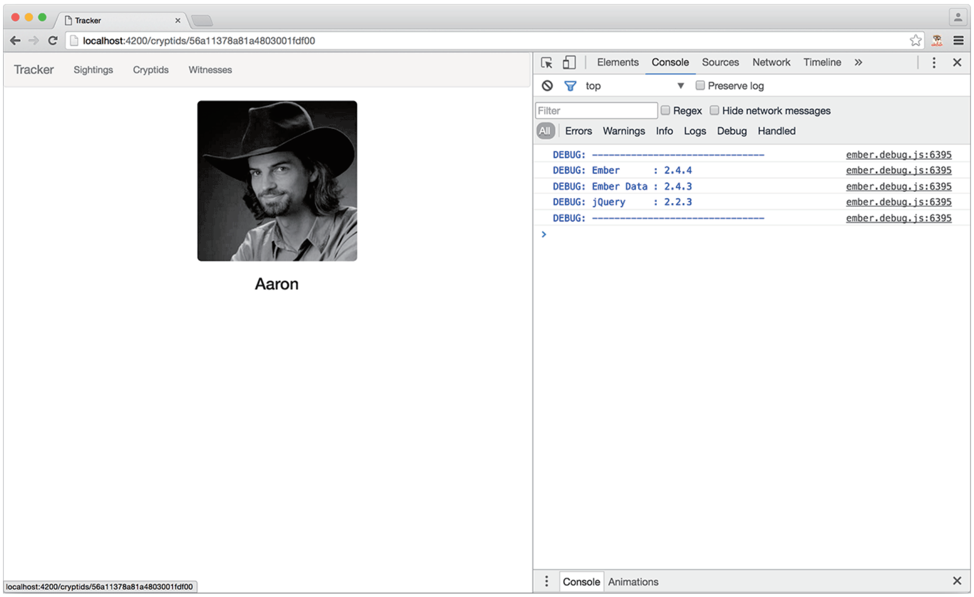


图23-6　神秘生物详情页

在后面的章节中，我们还会继续学习`{{#link-to}}`。记住一点，辅助方法是在模板渲染时运行的函数。Ember提供了一些辅助方法，但不代表只能使用这些辅助方法。

23.4　自定义辅助方法

在目击记录列表中，`sightedAt`字段以原始的日期字符串的形式展示，非常难看。为了让日期更加美观，再次使用在Chattrbox中用过的`moment`库对日期进行格式化。

从终端中安装moment：

```
bower install moment --save
```

修改ember-cli-build.js文件，使用app.import添加对moment库的引用：

```
...
  // 使用import引入资源文件
  app.import(bootstrapPath + 'javascripts/bootstrap.js');
  app.import('bower_components/moment/moment.js');

  return app.toTree();
};
```

由于修改了配置文件，所以需要重启服务器才能生效。在终端里使用Control＋C快捷键停止服务器，然后执行ember server重新启动它。

还需要生成一个辅助方法模块：

```
ember g helper moment-from
```

接下来的任务是创建一个函数，它接受日期作为参数，通过moment.js库对日期进行处理，最后返回一段HTML代码。另外，为返回的日期包裹上<span>标签，并在标签上加入Bootstrap提供的文本样式类。

打开刚刚生成的app/helpers/moment-from.js，添加这个新函数：

```
import Ember from 'ember';

export function momentFrom(params/*, hash*/) {
  return params;
}
export function momentFrom(params) {
  var time = window.moment(...params);
  var formatted = time.fromNow();
  return new Ember.Handlebars.SafeString(
    '<span class="text-primary">'
    + formatted + '</span>'
  );
}

export default Ember.Helper.helper(momentFrom);
```


现在已经定义好了辅助方法，接下来在app/templates/sightings/index.hbs模板中使用它：

```
...
      {{#if sighting.location}}
        <h3>{{sighting.location}} - {{sighting.sightedAt}}</h3>
        <p>{{moment-from sighting.sightedAt}}</p>
      {{else}}
...
```

辅助方法moment-from接受一个日期对象作为参数，返回日期格式化后的HTML字符串。在自定义辅助方法时，可以使用Ember提供的Ember.Handlebars.SafeString方法对字符串进行安全过滤。

看一下对日期进行格式化之后的效果（如图23-7所示）。

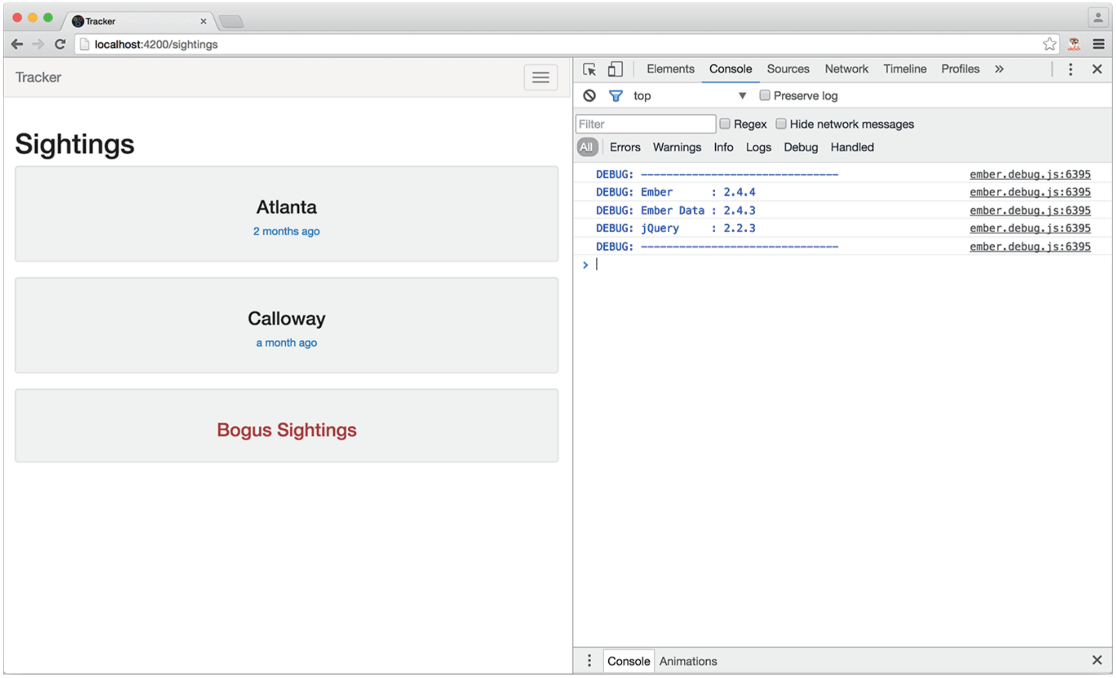


图23-7　使用moment.js格式化日期

moment.js把日期对象格式化成了“2 months ago”的样式，看起来是不是比“Tue Feb 9 2016 19:00:00 GMT-0500(EST)”好多了？（moment.js提供了许多针对格式化的参数，通过它的官网momentjs.com可以查看具体信息。）

通过定义辅助方法，可以将一些逻辑从模板中删除。定制辅助函数的好处是可以减少重复的代码，也可以对UI格式进行统一处理。这些抽象工作是创建Ember组件（Ember Component）的第一步，剩下的将会在第25章介绍。

在这一章中，你学会了如何在模板中输出模型的属性，以及如何通过条件语句和循环语句定制模板。另外，你还知道了如何对模板中HTML元素的属性赋值，以及如何创建包含动态参数的路由继而获取不同的数据并渲染到模板中。最后，你使用辅助方法创建了到路由的链接，此外还定制了格式化日期的辅助方法。

下一章会介绍应用生命周期中的最后一环——通过控制器创建和修改数据——以及动作、检索数据和创建装饰器。

23.5　初级挑战：为链接添加鼠标悬浮的内容

在鼠标移入时给链接展示一些额外的提示信息，可以通过为`{{#link-to}}`添加`title`属性

来实现，`title`的值是神秘生物的名字。

23.6　中级挑战：修改日期格式

`{{moment-from}}`使日期不再冗长，但是却过于精简，缺失了太多信息。请查阅`moment`的文档，尝试将输出样式改为“Sunday May 31, 2016”。

23.7　高级挑战：创建一个自定义缩略图辅助方法

在展示神秘生物的缩略图时使用了过多的代码。请创建一个用于展示缩略图的辅助方法，这个辅助方法需要一个参数：缩略图的路径。然后用这个辅助方法替换掉模板中渲染神秘生物图片的代码。

第 24 章

控制器

只剩MVC中的C没有介绍了。C代表**控制器**（controller），它的功能曾在第19章提过：负责应用的逻辑，获取模型实例并将其传递给视图。它还包含一些处理程序，可以对模型实例进行修改。

这一章要开发的控制器并不会包含太多代码。因为按照MVC的思想，一款复杂应用会被分解成多个部分，每个部分各司其职：模型负责管理数据，视图负责渲染界面，而控制器只负责**控制**模型和视图。

Ember会在运行时自动把控制器对象添加到应用中。有了控制器作为代理，模型数据才能在路由对象和模板之间传递。如果没有主动创建控制器，Ember会认为应用只需要把模型数据传给模板，它会自动创建一个这样的控制器。

通过创建控制器可以实现许多功能，例如可以在其中监听事件或动作。也可以在控制器中定义装饰器，对要展示的数据进行调整而又不影响模型中的数据。

Tracker应用的主要功能之一是让用户自主添加目击记录。为了实现这个功能，需要创建一个路由和一个控制器，并在控制器中添加相关的属性和动作。

前面已经创建了新建目击记录的路由app/routes/sightings/new.js，现在需要在这个页面中展示一个表单，表单中要有神秘生物和目击者的列表供用户选择，以便创建新的目击记录。每一条目击记录都会关联到一种神秘生物和一位或多位目击者。这个表单如图24-1所示。

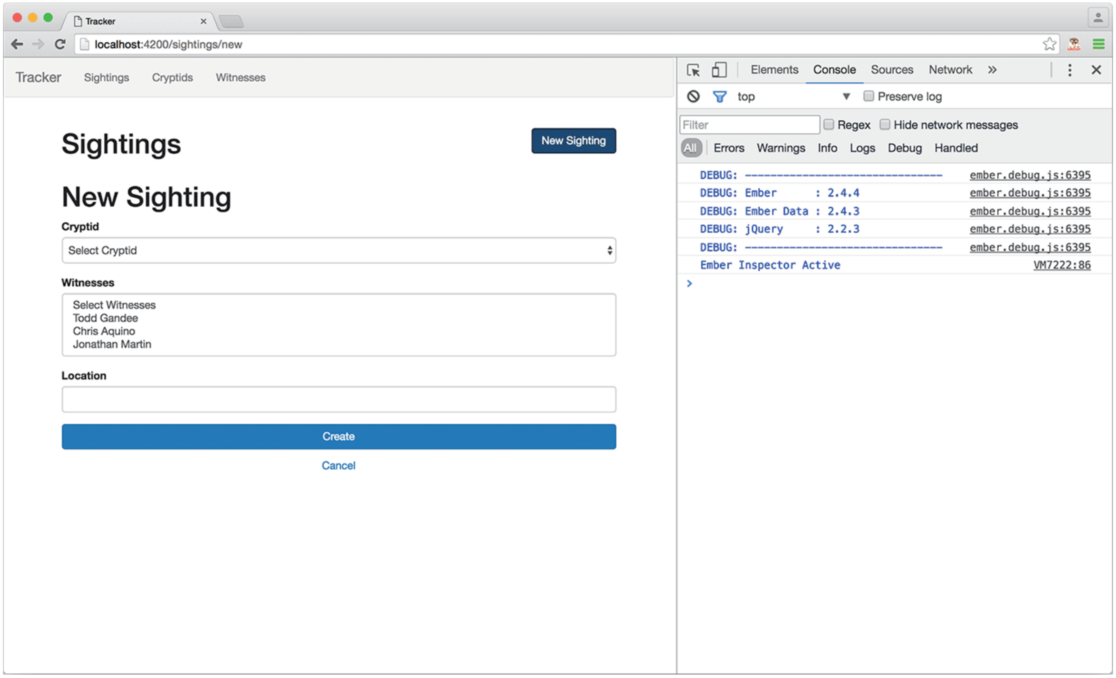


图24-1 新建目击记录的表单

准备工作完成后，就可以着手创建用于管理表单中各种事件的控制器了。除此之外，还需要提供修改和删除已有目击记录的功能。

24.1 新建目击记录

之前创建的目击记录的模型会返回全部目击记录数据，而新建记录的模型则返回新创建记录的结果。此外，还要返回一系列神秘生物和目击者的Promise对象的集合，在这儿用的是Ember.RSVP.hash({})。

打开app/routes/sightings/new.js，在其中添加model钩子，返回值是用Ember.RSVP.hash处理的Promise的集合：

```
...
export default Ember.Route.extend({
  model() {
    return Ember.RSVP.hash({
      sighting: this.store.createRecord('sighting')
    });
  }
});
```

当这条路由被访问时，一条新的目击记录会被返回。如果这时查看记录列表，就能看到一条

空白记录，这正是我们刚刚添加的记录。由于只在本地创建了这条数据而没有将其同步到服务器端，所以它是一条**脏数据**，在本章最后会处理脏数据的问题。就目前来说，只需要了解createRecord在本地创建了一条新的目击记录就可以了。

在新建记录时，用到了神秘生物列表和目击者列表。这里用Ember.RSVP.hash({})返回一个Promise的对象。Ember.RSVP.hash({})的参数对象中只包含一个属性sighting，它代表在渲染模板时需要用到model.sighting的数据，也就是刚刚创建的目击记录。

还需要在模型中加入对神秘生物和目击者的检索（别忘了this.store.createRecord('sighting')后面的逗号）：

```
...
export default Ember.Route.extend({
  model() {
    return Ember.RSVP.hash({
      sighting: this.store.createRecord('sighting'),
      cryptids: this.store.findAll('cryptid'),
      witnesses: this.store.findAll('witness')
    });
  }
}
```

在新建记录的模板中使用<select>标签，为用户提供神秘生物列表和目击者列表。在动手修改模板之前，先来介绍一个Ember CLI新插件，它比原生的<select>标签更简单易用。

在终端中安装emberx-select：

```
ember install emberx-select
```

这个插件的名称是x-select，在模板中使用它就能省去在<select>标签上绑定onchange事件的操作。

为了使用刚安装的x-select，需要先重启ember服务器。

准备工作现已就绪，开始编辑模板。修改app/templates/sightings/new.hbs，在其中添加新建目击记录的表单：

```
<h1>New Route</h1>
<h1>New Sighting</h1>
<form>
  <div class="form-group">
    <label for="name">Cryptid</label>
    {{#x-select value=model.sighting.cryptid class="form-control"}}
      {{#x-option}}Select Cryptid{{/x-option}}
      {{#each model.cryptids as |cryptid|}}
        {{#x-option value=cryptid}}{{cryptid.name}}{{/x-option}}
      {{/each}}
    {{/x-select}}
  </div>
  <div class="form-group">
    <label>Witnesses</label>
    {{#x-select value=model.sighting.witnesses multiple=true class="form-control"}}
      {{#x-option}}Select Witnesses{{/x-option}}
      {{#each model.witnesses as |witness|}}
```

```
        {{#x-option value=witness}}{{witness.fullName}}{{/x-option}}
      {{/each}}
    {{/x-select}}
  </div>
  <div class="form-group">
    <label for="location">Location</label> {{input value=model.sighting.location
      type="text" class="form-control" name="location" required=true}}
  </div>
</form>
```

哇！这段代码用到了之前学过的所有内容，还涉及新知识。路由用到了一个新的Ember.RSVP方法，模板用到了辅助函数，还学习了新组件{{x-select}}和{{x-option}}。

{{x-select}}组件的本质是构建了一个<select>元素，让用户以选择的方式为属性赋值。在使用方式上，它与<select>差别不大，在赋值时使用的是Ember提供的数据绑定。这里使用model.sighting.cryptid为{{x-select}}的value属性赋值。当用户选择新的列表项时，组件会对onchange事件作出响应。这么做可行是因为cryptid属性需要使用神秘生物的记录作为它的值。

在创建目击者列表时，需要额外添加一个属性multiple=true，它的作用是允许用户选择多个目击者。选择的结果会转换成使用hasMany描述的目击者的集合。

此外还需要在目击记录列表页面添加一个按钮，链接到新建记录页面。修改app/templates/sightings.hbs，在其中添加按钮：

```
<h1>Sightings</h1>
<div class="row">
  <div class="col-xs-6">
    <h1>Sightings</h1>
  </div>
  <div class="col-xs-6 h1">
    {{#link-to "sightings.new" class="pull-right btn btn-primary"}}
      New Sighting
    {{/link-to}}
  </div>
</div>
{{outlet}}
```

这里利用Bootstrap提供的样式创建了一个按钮，访问http://localhost:4200/sightings查看效果（如图24-2所示）。

新建记录的页面现在有了入口，在所有与目击记录相关的页面中都能看到这个按钮，点击它就可以创建目击记录。

请注意，当处在sightings.new页面时，新建目击记录的按钮是高亮的，这是因为Ember会自动为指向当前路由的链接添加active类名。这么做的目的是给用户视觉上的提示，便于用户区分指向当前页面的是哪个链接。

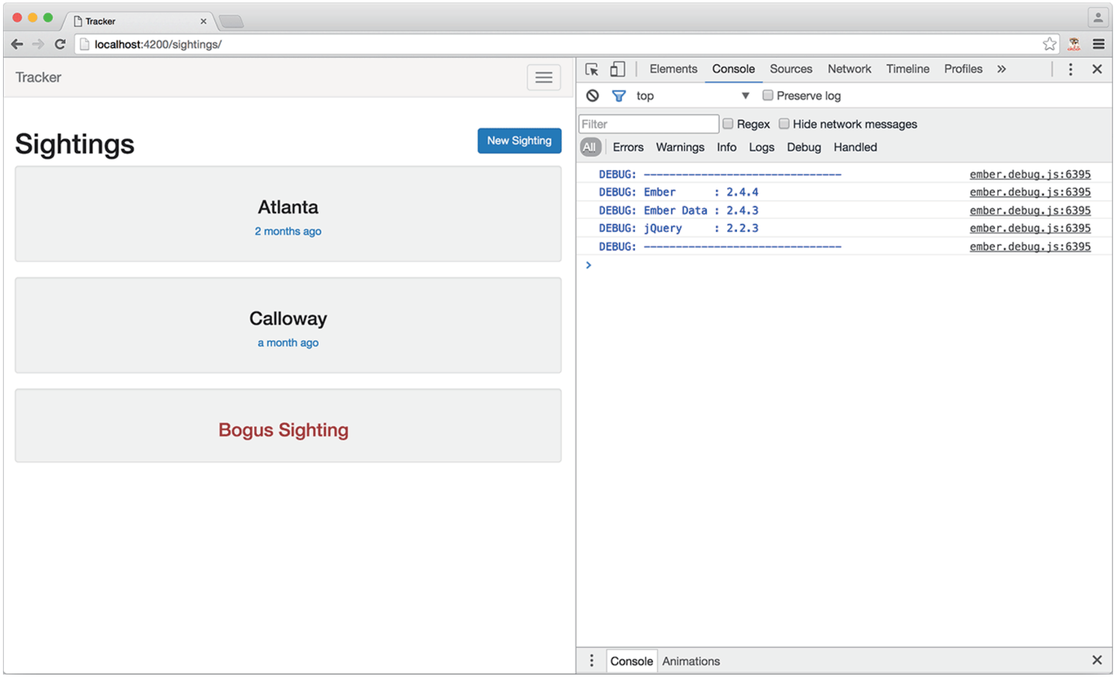


图24-2　新建目击记录按钮

在应用中，动作是处理表单事件和其他各类事件的核心。`actions`属性是一个对象，包含了许多事件名称到方法的映射，在模板中需要通过这些名称来调用对应的方法。

现在来着手创建sightings.new路由的控制器，在命令行中执行：

```
ember g controller sightings/new
```

Ember生成了app/controllers/sightings/new.js文件。打开文件，添加`create`和`cancel`两个动作，在创建目击记录时需要用到它们：

```
import Ember from 'ember';

export default Ember.Controller.extend({
  actions: {
    create() {
    },
    cancel() {
    }
  }
});
```

一般情况下，需要为表单元素设置一个`action`属性，当表单提交时会向`action`指定的URL发送请求。但对于Ember应用来说，只需要为表单指定一个动作名称，这个动作就会在表单提交时触发。

修改app/templates/sightings/new.hbs，为表单元素添加用于提交的动作。同时也在表单的后面添加Create和Cancel这两个按钮：

```
<h1>New Sighting</h1>
<form {{action "create" on="submit"}}>
...
  <div class="form-group">
    <label for="location">Location</label> {{input value=model.location
      type="text" class="form-control" name="location" required=true}}
  </div>
  <button type="submit" class="btn btn-primary btn-block">Create</button>
  <button {{action 'cancel'}} class="btn btn-link btn-block">Cancel</button>
</form>
```

正常情况下，Atom编辑器无法识别`{{action}}`这种语法，所以它会报错，这种错误可以忽略。当然，也可以为Atom安装Language-Mustache扩展包。在设置中启用后，Atom就可以识别这种语法了。包的地址是atom.io/packages/language-mustache。

在调用`{{action}}`时传入了字符串类型的参数，与app/controllers/sightings/new.js中创建的动作相对应，这些动作都绑定了各自的事件处理程序。`{{action}}`的另一个参数是`on`，它代表触发这个动作的操作类型。默认是通过点击触发，如果`on`为空则使用默认值。

对表单中的`{{action}}`使用`on="submit"`，表示在表单提交时触发动作。不对取消按钮上的`{{action}}`添加`on`属性，也就意味着使用默认值，在点击操作时触发动作。

控制器中定义的动作已经能够实现表单的提交或取消，但这些动作的实现代码目前还是空的。现在来创建这些动作，用下面的代码替换app/controllers/sightings/new.js中的对应内容，保存后返回目击记录列表页面：

```
...
  actions: {
    create() {
      var self = this;
      this.get('model.sighting').save().then(function() {
        self.transitionToRoute('sightings');
      });
    },
    cancel() {
    }
  }
});
```

`create`方法会在表单提交时被调用。首先在这个方法中创建`self`变量指向控制器自身，接着获取sighting模型并调用`save`。

最后一步是把数据保存到服务端。每个模型对象上都有一个`hasDirtyAttributes`属性，它会在保存完模型后被设置为`false`。

模型在保存时会返回一个`Promise`对象。调用`Promise`的`then`方法，传入一个回调函数，这个回调函数会在模型保存成功后调用。最后使用`transitionToRoute`返回目击记录列表页。

访问http://localhost:4200/sightings/new，查看创建的表单（如图24-3所示）。

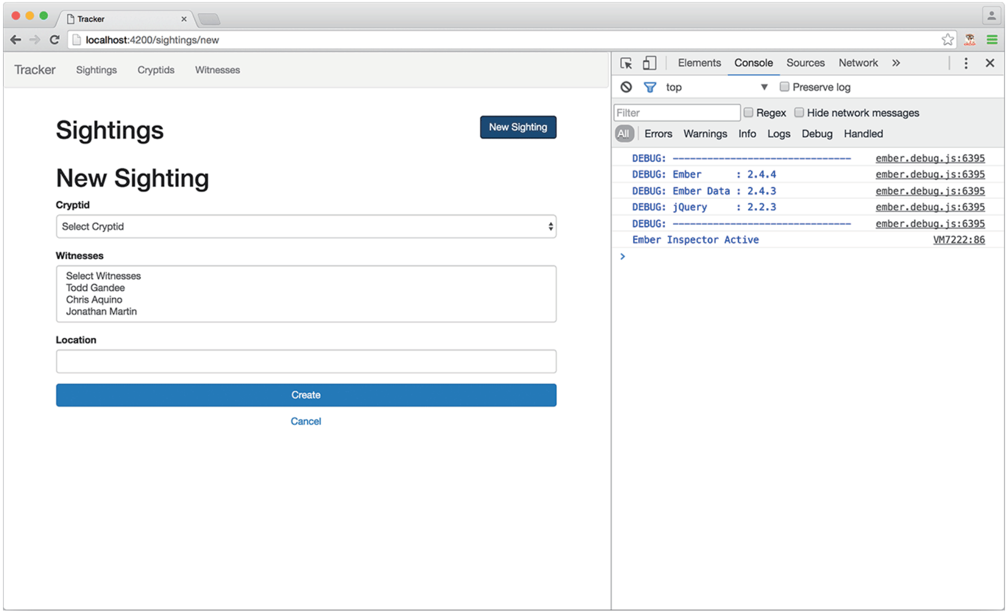


图24-3　新建目击记录表单

填写表单后点击Create。虽然已经添加了一条新记录，但是在这里看到的还是之前的模拟数据。修改app/routes/sightings.js，删除模拟数据，换成从服务器端获取的目击记录数据。

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    let record1 = this.store.createRecord('sighting', {
      location: 'Atlanta',
      sightedAt: new Date('2016-02-09')
     });
     record1.set('location', 'Paris, France');
     console.log("Record 1 location: " + record1.get('location'));

    let record2 = this.store.createRecord('sighting', {
      location: 'Calloway',
      sightedAt: new Date('2016-03-14')
    });

    let record3 = this.store.createRecord('sighting', {
      location: '',
      sightedAt: new Date('2016-03-21')
    });

    return [record1, record2, record3];
    return this.store.findAll('sighting', {reload: true});
  }
});
```

现在，应用已经具备了创建和检索的功能。

注意findAll方法的第二个参数，它是一个配置对象，其中只包含了reload一个属性。这个参数告诉store在每次调用路由模型时都从API请求最新的数据。添加这个参数表明在每次加载列表时都希望展示最新的数据。

接下来处理cancel动作，在执行cancel时需要删除内存中目击记录实例的脏数据。和第21章一样，这里也使用model.deleteRecord方法删除数据，把它添加到app/controllers/sightings/new.js中：

```
...
  actions: {
    create() {
      var self = this;
      this.get('model.sighting').save().then(function() {
        self.transitionToRoute('sightings');
      });
    },
    cancel() {
      this.get('model.sighting').deleteRecord();
      this.transitionToRoute('sightings');
    }
  }
...
```

删除操作完成后，用户会回到列表页。这里使用的方案只针对用户点击取消按钮的情况，但是如果用户使用导航条直接跳转到列表页或其他页面呢？

如果这条脏数据没有被删除，那么直到用户关闭浏览器为止，它都会一直保存在内存中。为了能及时删除这条脏数据，需要在路由中添加一个动作。

在路由的生命周期中，有一些预定的动作会因路由的特定状态或路由状态的转变而触发。可以通过覆盖这些动作，实现在路由变换时执行指定的操作。

修改app/routes/sightings/new.js，添加willTransition动作用于删除脏数据：

```
...
  model(){
    return Ember.RSVP.hash({
      sighting: this.store.createRecord('sighting'),
      cryptids: this.store.findAll('cryptid'),
      witnesses: this.store.findAll('witness')
    });
  },
  actions: {
    willTransition() {
      var sighting = this.get('controller.model.sighting');
      if(sighting.get('hasDirtyAttributes')){
        sighting.deleteRecord();
      }
    }
  }
});
```

当路由改变时会触发`willTransition`。在这个方法中，首先判断`hasDirtyAttributes`是否为`true`，如果为`true`则会调用`deleteRecord`销毁模型对象。

这些修改解决了创建记录时的脏数据问题，同时只对控制器做了最小的修改就把数据保存到服务端。

24.2　编辑目击记录

在创建（create）和读取（read）数据之后，按照CRUD的顺序，下一步应该更新（update）数据。虽然已经创建了更新数据的路由，但是还没有添加编辑入口。接下来在目击记录的列表中添加编辑按钮。同时，要在编辑记录的模板中添加一个表单，其中包含目击记录相关的字段。此外，修改编辑目击记录的模型，增加对目击者、神秘生物和目击记录的检索。修改app/router.js，在编辑目击记录的路由上加入动态参数。最后，创建一个控制器为表单添加动作。

打开app/templates/sightings/index.hbs，在列表中添加编辑按钮。为了使列表页更加丰富，还要添加神秘生物的名称和图片。

```
...
      <div class="media well">
        <img class="media-object thumbnail" src="{{if sighting.cryptid.profileImg
          sighting.cryptid.profileImg 'assets/images/cryptids/blank_th.png'}}"
          alt="{{sighting.cryptid.name}}" width="100%" height="100%">
        <div class="caption">
          <h3>{{sighting.cryptid.name}}</h3>
          {{#if sighting.location}}
            <h3>{{sighting.location}}</h3>
            <p>{{moment-from sighting.sightedAt}}</p>
          {{else}}
            <h3 class="text-danger">Bogus Sighting</h3>
          {{/if}}
        </div>
        {{#link-to 'sighting.edit' sighting.id tagName="button"
          class="btn btn-success btn-block"}}
          Edit
        {{/link-to}}
      </div>
...
```

访问http://localhost:4200/sightings，查看编辑按钮（如图24-4所示）。

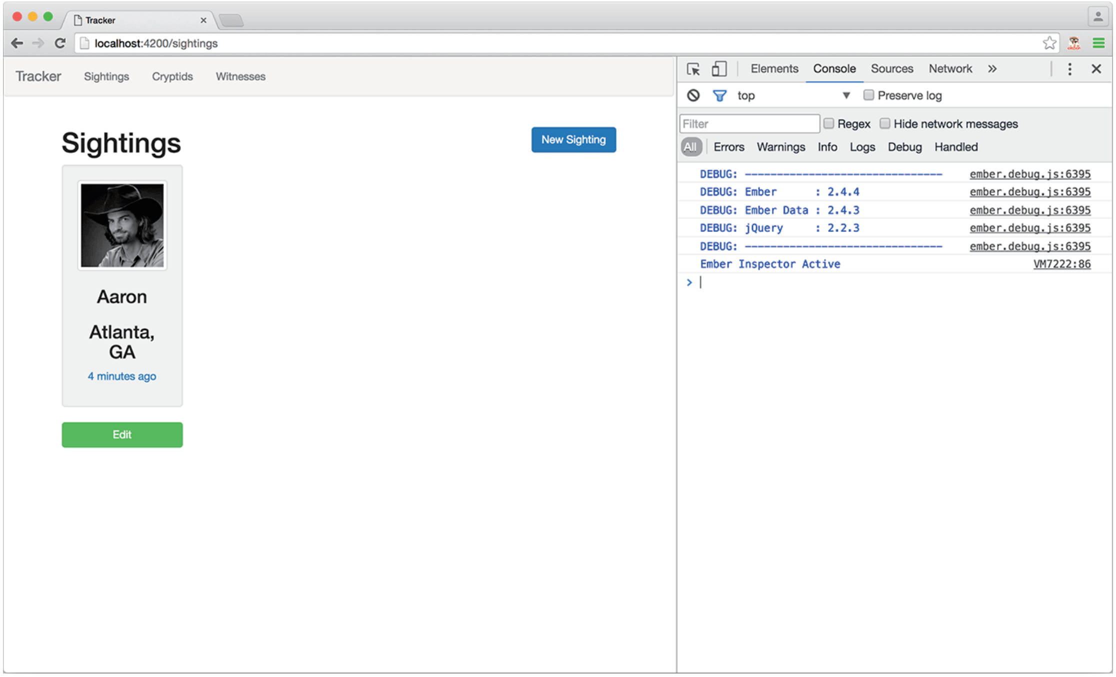


图24-4　带有编辑按钮的列表页

接下来修改router.js，为编辑目击记录的路由添加动态参数。

```
...
  this.route('sighting', function() {
    this.route('edit', {path: "sightings/:sighting_id/edit"});
  });
...
```

接下来修改app/routes/sightings/edit.js，添加对目击记录、神秘生物和目击者的检索方法：

```
...
export default Ember.Route.extend({
  model(params) {
    return Ember.RSVP.hash({
      sighting: this.store.findRecord('sighting', params.sighting_id),
      cryptids: this.store.findAll('cryptid'),
      witnesses: this.store.findAll('witness')
    });
  }
});
```

接下来修改app/templates/sightings/edit.hbs，在模板中加入编辑记录的表单，这个表单和新建记录的表单几乎没有差别。

```
{{outlet}}
<h1>Edit Sighting:
  <small>
```

```
      {{model.sighting.location}} -
       {{moment-from model.sighting.sightedAt}}
    </small>
  </h1>
  <form {{action "update" model on="submit"}}>
    <div class="form-group">
      <label for="name">Cryptid</label>
      {{input value=model.sighting.cryptid.name type="text" class="form-control"
        name="location" disabled=true}}
    </div>
    <div class="form-group">
      <label>Witnesses</label>
      {{#each model.sighting.witnesses as |witness|}}
        {{input value=witness.fullName type="text" class="form-control"
          name="location" disabled=true}}
      {{/each}}
    </div>
    <div class="form-group">
      <label for="location">Location</label>
      {{input value=model.sighting.location type="text" class="form-control"
        name="location" required=true}}
    </div>
    <button type="submit" class="btn btn-info btn-block">Update</button>
    <button {{action 'cancel'}} class="btn btn-block">Cancel</button>
  </form>
```

现在只剩下控制器没有完成了。在控制器中创建表单动作的响应方法。另外，由于目前只能修改地址，所以要暂时禁用神秘生物和目击者字段。

创建控制器：

```
ember g controller sighting/edit
```

打开刚生成的文件app/controllers/sightings/edit.js，添加`update`和`cancel`两个动作：

```
import Ember from 'ember';

export default Ember.Controller.extend({
  sighting: Ember.computed.alias('model.sighting'),
  actions: {
    update() {
      if(this.get('sighting').get('hasDirtyAttributes')){
        this.get('sighting').save().then(() => {
          this.transitionToRoute('sightings');
        });
      }
    },
    cancel() {
      if(this.get('sighting').get('hasDirtyAttributes')){
        this.get('sighting').rollbackAttributes();
      }
      this.transitionToRoute('sightings');
    }
  }
});
```

和创建记录时一样，提交修改时也只需要调用save。Ember会自动进行判断，只有在记录发生改变时，即sighting.get('hasDirtyAttributes')为true时才向API发起请求。

注意一下这里的Ember.computed.alias，它的功能是为属性创建别名，可用于任何属性，尤其对嵌套比较深的属性特别有用。通过这个计算属性，我们在使用sighting时可以少写许多代码。

24.3　删除目击记录

时不时就会有一些目击记录被证实是骗局，虽然并不多，但确实需要一种用来删除虚假或者陈旧数据的方法。回想一下，在第21章中曾使用record.destroyRecord删除过记录。

首先在app/templates/sightings/edit.hbs中添加一个删除按钮。

```
  <button type="submit" class="btn btn-info btn-block">Update</button>
  <button {{action 'cancel'}} class="btn btn-block">Cancel</button>
</form>

<hr>
<button {{action 'delete'}} class="btn btn-block btn-danger">
  Delete
</button>
```

除了按钮外，再添加一条水平线，作用是把删除按钮和表单中的更新和取消按钮分隔开。这里的<hr>是一个有效的视觉工具，它可以明确地提醒用户删除操作与编辑操作有所区别。

接下来，在app/controllers/sightings/edit.js中添加删除动作：

```
...
  cancel() {
    if(this.get('sighting').get('hasDirtyAttributes')){
      this.get('sighting').rollbackAttributes();
    }
    this.transitionToRoute('sightings');
  },
  delete() {
    var self = this;
    if (window.confirm("Are you sure you want to delete this sighting?")) {
      this.get('sighting').destroyRecord().then(() => {
        self.transitionToRoute('sightings');
      });
    }
   }
  }
});
```

删除动作使用了window.confirm让用户对操作进行确认。除了多一个判断语句外，删除动作与其他动作的逻辑并无不同：获取模型，调用方法，当API请求完成时执行异步回调。

打开http://localhost:4200/sightings，任选一条记录，点击编辑按钮。新打开的页面即为app/templates/sightings/edit.hbs模板渲染后的效果，其中就能看到新添加的删除按钮（如图24-5所示）。

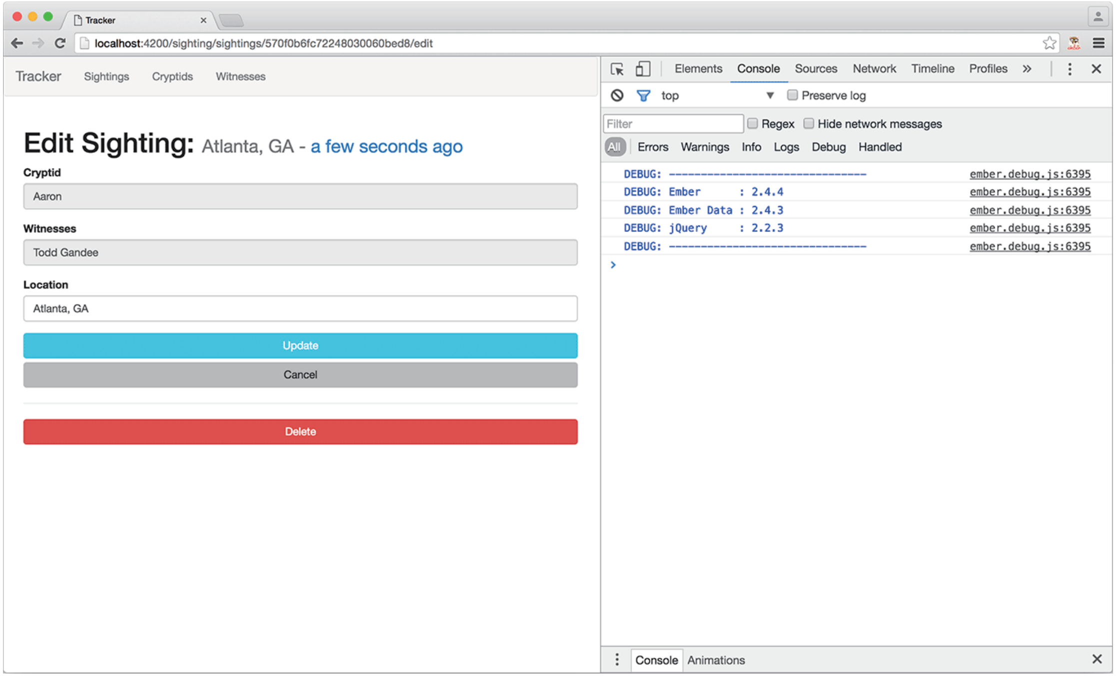


图24-5　编辑目击记录的表单

从创建到销毁，与记录相关的各种操作都已经完成了。

24.4　路由动作

动作并不是控制器独有的。路由也可以为模板定义动作，还可以重写生命周期里的动作。调用一个动作时，它会沿着模板–控制器–路由–父级路由的路径一路向上。

如果不单独定义控制器，路由也可以代替控制器，实现它的功能。这一点似乎与前面讨论的应用逻辑分离相互矛盾，但其实Ember把控制器的工作分为了两个部分：路由信息和控制器逻辑。有时候会在路由中包含较多逻辑，而有时候会在控制器中包含较多逻辑。应用中可能有一些逻辑被分割成小块代码，但同时也有一些文件中堆积了多个动作和装饰器。

为了弄明白路由如何充当控制器，把`create`和`cancel`事件的定义从app/controllers/sightings/new.js移动到app/routes/sightings/new.js。通过这样的修改，也能从另一个角度理解这些方法和对象。而且如果只想使用路由和组件（下一章会介绍）来控制整个应用的视图，这也是个好方案。

首先，在app/routes/sightings/new.js中添加动作：

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
  ...
```

```
  },
  sighting: Ember.computed.alias('controller.model.sighting'),
  actions: {
    willTransition() {
      var sighting = this.get('controller.model.sighting');
      if(sighting.get('hasDirtyAttributes')) {
        sighting.deleteRecord();
      }
    },
    create() {
      var self = this;
      this.get('sighting').save().then(function(data) {
        self.transitionTo('sightings');
      });
    },
    cancel() {
      this.get('sighting').deleteRecord();
      this.transitionToRoute('sightings');
    }
  }
});
```

这段代码也使用Ember.computed.alias为sighting对象创建了一个别名。不同的是，当从路由中获取控制器的属性时你需要明确这个对象存在的位置，因为路由中的model和控制器中的model并不是同一个对象。为了能从路由中获取控制器的sighting属性，需要使用get('controller.model.sighting')。为了减少冗余代码，可以为它起一个别名。

然后，从app/controllers/sightings/new.js中删除动作：

```
import Ember from 'ember';

export default Ember.Controller.extend({
  actions: {
    create() {
      var self = this;
      this.get('model.sighting').save().then(function() {
        self.transitionToRoute('sightings');
      });
    },
    cancel() {
      this.get('model.sighting').deleteRecord();
      this.transitionToRoute('sightings');
    }
  }
});
```


请确保修改后的文件已经重新加载（或者ember服务器已经重启）。访问http://localhost:4200/sightings/new，在其中添加一条新记录以确保路由中的动作可以正常工作。

这时app/controllers/sightings/new.js文件已经有些多余了，可以删掉它，使目录变得清爽一些。虽然删除了控制器文件，但在运行时Ember仍会为应用创建默认的控制器对象。

这一章重点关注了Ember的控制器，学习了怎样为模板创建动作和属性，怎样在保存数据后

跳转到新页面，以及怎样在取消保存或切换到其他页面时销毁记录。通过使用动作，可以用简单的回调函数完成对模型数据的修改。另外，使用控制器可以设置页面独有属性，而不需要向模型数据添加关系。最后，还完成了对目击记录的更新和删除，实现了完整的CRUD操作。

出于快速开发的考量，Ember不强制开发者使用控制器，而是允许将一些控制器的细节在路由中实现。此外，还可以通过控制器来微调视图和控制模型数据。动作是应用与用户交互的核心，而且动作还可以存在于路由、控制器和组件中。组件会在下一章（也是最后一章）进行讲解。

24.5　初级挑战：目击记录详情页

请创建一个目击记录的详情页面，相关的文件是app/templates/sightings/index.hbs和app/routes/sightings.js。在这个页面中展示神秘生物的图片、目击地点以及目击者的列表，要是能加上编辑按钮就更好了。（提示：可以直接在路由中添加动作。）

24.6　中级挑战：目击日期

在创建和编辑目击记录时，给控制器中添加一个`sightingDate`属性，在模板中创建对应的表单元素与该属性进行绑定。在添加日期输入框时，既可以简单地使用纯文本，也可以使用`input[type="date"]`。用`moment`库将获取到的日期转化成ISO8601规定的日期格式，然后赋给目击记录的`sightingAt`属性。

24.7　高级挑战：添加和删除目击者

在创建目击记录时，根据用户在`<select>`元素中点击触发的`onchange`事件，构造出对应的目击者列表。请创建一个新属性用于临时保存目击者列表。在用户提交表单时，将这个属性的值赋给目击记录的`witnesses`属性。

另外请在页面中展示已选的目击者列表，同时为其中的每个选项添加一个删除按钮。当一个目击者被用户选择后，需要把它从`<select>`的选项中移除。可以用这两个动作：`addWitness`和`removeWitness`。

第 25 章 组　　件

在Ember中，组件是包含视图和控制器属性的对象。组件背后的理念是为可重用的DOM元素创建独立的作用域或者上下文环境。组件会包含一些属性，通过这些属性可以定制组件输出的内容和样式。另外，组件还可以通过属性接受从父控制器或父级路由传来的动作（如图25-1所示）。

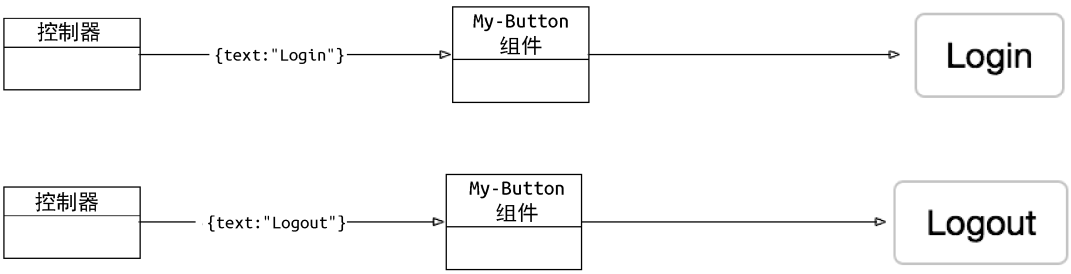


图25-1　组件属性

回过头来看整个应用，你会发现许多代码片段都出现在多个模板中，而且相互之间差异不大。如果一个片段可以从模板中抽取出来并通过变量来描述状态，那么这个代码片段就很适合转化成一个组件。

本章会提供一些简单的示例，展示怎样将DOM元素包装成JavaScript对象（组件），然后把它用在多个页面中。组件的逻辑有些类似于第23章的辅助方法，它们以标签属性的形式接受参数，然后经过一些处理最终输出HTML代码。组件也拥有自己的动作和属性，可以通过与用户交互更新自己的状态。

本章内容只是开发可扩展应用的冰山一角。在现实中一款复杂的应用可能拥有上百个页面，为了提高应用界面的一致性和代码的可维护性，Ember会高度依赖组件。本章中的例子只是现实项目的一个缩影，但它们的开发思路是一致的：将模板的内容替换成可重用的页面和单个元素。

25.1　迭代器组件

通常可以把`{{#each}}`迭代渲染的一组元素转换成组件，这也是把代码片段转化成组件的一

个典型案例。迭代器可能在每一次迭代时都需要渲染一个<div>作为容器，然后根据被迭代对象的属性在容器中渲染标题、图片路径、按钮并调整元素的样式。为了能重用这些代码，可以将它们包装成组件模板。此外还需要创建一个组件JavaScript文件。组件文件有些类似于控制器，可以在其中创建装饰器和动作的响应方法。

目击记录列表就是用{{#each}}渲染的，要创建的第一个组件就是这个列表的单个元素。首先通Ember CLI创建一个组件，在终端中运行：

```
ember g component listing-item
```

这条命令会创建三个文件：app/components/listing-item.js、app/templates/components/listing-item.hbs和tests/integration/components/listing-item-test.js（用于测试）。

在创建好组件后，找出需要替换成组件的代码片段。打开app/templates/sightings/index.hbs模板，下面这段代码中的一部分将被移动到组件模板中，先大致浏览一下：

```
<div class="row">
  {{#each model as |sighting|}}
    <div class="col-xs-12 col-sm-3 text-center">
      <div class="media well">
        <img class="media-object thumbnail" src="{{if sighting.cryptid.profileImg
          sighting.cryptid.profileImg 'assets/images/cryptids/blank_th.png'}}"
          alt="{{sighting.cryptid.name}}" width="100%" height="100%">
        <div class="caption">
          <h3>{{sighting.cryptid.name}}</h3>
          {{#if sighting.location}}
            <h3>{{sighting.location}}</h3>
            <p>{{moment-from sighting.sightedAt}}</p>
          {{else}}
            <h3 class="text-danger">Bogus Sighting</h3>
          {{/if}}
        </div>
        {{#link-to 'sighting.edit' sighting.id tagName="button"
          class="btn btn-success btn-block"}}
          Edit
        {{/link-to}}
      </div>
    </div>
  {{/each}}
</div>
```

这段代码中的容器<div>（阴影标注的）使用了类名col-xs-12，也就是说元素的尺寸是固定的，也是为当前页面定制的。如果直接把它用在组件中，那么整个应用中所有使用该组件的位置都将展示同样尺寸的列表。

不过可以将<div class="media well">容器及其内容迁移到组件中，这样组件就可以适用于各种大小的容器，而单个列表条目的结构并不会发生变化。组件中包含的是列表条目的主要元素，包括media容器、图片、标题和编辑按钮。

打开app/templates/components/listing-item.hbs，在其中添加神秘生物的图片和名称：

```
<img class="media-object thumbnail" src="{{imagePath}}" alt="{{name}}"
  width="100%" height="100%">
<div class="caption">
  <h3>{{name}}</h3>
  {{yield}}
</div>
```

从外部来看，一个组件就像一个独立DOM元素，组件模板中所有的子元素也就是这个DOM元素的子元素。在默认情况下，Ember的HTMLBars引擎会通过JavaScript创建一个`<div>`元素作为容器，然后在这个容器中渲染组件模板。

上面的代码包含一张图片和一个标题元素，它们会被添加到由`{{#listing-item}}`创建的`<div>`容器中。和普通的模板一样，在组件的模板中也可以渲染变量等一些动态内容（先忽略代码中的`{{yield}}`，稍后介绍它）。

把组件添加到页面模板中，它会渲染出如下内容：

```
<div>
  <img class="media-object thumbnail" src="[cryptid's imagePath string]"
    alt="[cryptid's name string]" width="100%" height="100%">
  <div class="caption">
    <h3>[cryptid's name string]</h3>
    {{yield}}
  </div>
</div>
```

模板中动态的部分（例如`{{name}}`和`{{imagePath}}`）通过属性的形式传入组件。`{{yield}}`用于在它所处的位置渲染传入组件的子元素。组件的模板文件用于布局或展现主要元素，而`{{yield}}`则用于在每个组件实例中渲染不同的子元素。稍后就会用到`{{yield}}`。

尽管最终不会把app/templates/sightings/index.hbs中的内容替换成上面的标签，但是这样更好理解。

将app/templates/sightings/index.hbs中的内容换成组件：

```
<div class="row">
  {{#each model as |sighting|}}
    <div class="col-xs-12 col-sm-3 text-center">
      <div class="media well">
        <img class="media-object thumbnail" src="{{if sighting.cryptid.profileImg
          sighting.cryptid.profileImg 'assets/images/cryptids/blank_th.png'}}"
          alt="{{sighting.cryptid.name}}" width="100%" height="100%">
        <div class="caption">
          <h3>{{sighting.cryptid.name}}</h3>
          {{#if sighting.location}}
            <h3>{{sighting.location}}</h3>
            <p>{{moment-from sighting.sightedAt}}</p>
          {{else}}
            <h3 class="text-danger">Bogus Sighting</h3>
          {{/if}}
        </div>
        {{#link-to 'sighting.edit' sighting.id tagName="button"
          class="btn btn-success btn-block"}}
```

25

```
        Edit
      {{/link-to}}
    </div>
    {{#listing-item imagePath=sighting.cryptid.profileImg
      name=sighting.cryptid.name}}
    {{/listing-item}}
  </div>
{{/each}}
</div>
```

这里只用了一行代码就替换了之前的大段代码。虽然目前功能还不完备，不过很快就会补上。接下来把组件的容器上缺失的类名"media"和"well"补回来。

修改app/components/listing-item.js，为组件添加classNames属性：

```
import Ember from 'ember';

export default Ember.Component.extend({
  classNames: ["media", "well"]
});
```

在组件创建<div>容器时，会取这里的classNames属性的值作为容器的类名。

修改后的组件会渲染出如下代码：

```
<div class="media well">
  <img class="media-object thumbnail" src="[imagePath string]" alt="[name string]"
    width="100%" height="100%">
  <div class="caption">
    <h3>[name string]</h3>
    {{yield}}
  </div>
</div>
```

接下来要为组件添加子元素，这些元素会被渲染在组件模板中{{yield}}的位置。当不同页面调用同一组件时，就可以在组件中展示不同的内容。打开app/templates/sightings/index.hbs文件，添加下面的代码，它们会被渲染到{{yield}}所在的位置。

```
...
      {{#listing-item imagePath=sighting.cryptid.profileImg
        name=sighting.cryptid.name}}
        {{#if sighting.location}}
          <h3>{{sighting.location}}</h3>
            <p>{{moment-from sighting.sightedAt}}</p>
        {{else}}
          <h3 class="text-danger">Bogus Sighting</h3>
        {{/if}}
        {{#link-to 'sighting.edit' sighting.id tagName="button"
          class="btn btn-success btn-block"}}
          Edit
        {{/link-to}}
      {{/listing-item}}
```

这段代码你应该不陌生，它会为目击记录列表添加缺失的位置信息、时间信息和编辑按钮。

25.2　“拧干”组件的“水分”

现在是时候让你的代码变“DRY”了。

什么意思？DRY，不是真的变“干”，而是Don’t Repeat Yourself（不要重复你自己）的缩写。这是一种编码原则，简单来说就是同样的逻辑在一份代码中只出现一次。

回想一下，目击记录列表和神秘生物列表都包含了一个class="media well"的容器，在容器中都有图片和标题。所以这些代码是可以进行DRY优化的。虽然这两个列表结构并不完全相同，但是可以通过{{yield}}解决。

首先将刚刚创建的{{#listing-item}}组件添加到神秘生物列表里。打开app/templates/cryptids.hbs模板，将<div class="media well">元素和它的子元素替换成如下代码：

```
<div class="row">
  {{#each model as |cryptid|}}
    <div class="col-xs-12 col-sm-3 text-center">
      <div class="media well">
        {{#link-to 'cryptid' cryptid.id}}
          <img class="media-object thumbnail"
            src="{{if cryptid.profileImg cryptid.profileImg
            'assets/images/cryptids/blank_th.png'}}"
            alt="{{cryptid.name}}" width="100%" height="100%">
        {{/link-to}}
        <div class="caption">
          <h3>{{cryptid.name}}</h3>
        </div>
      </div>
      {{#link-to 'cryptid' cryptid.id}}
        {{listing-item imagePath=cryptid.profileImg name=cryptid.name}}
      {{/link-to}}
    </div>
  {{else}}
    <div class="jumbotron">
      <h1>No Creatures</h1>
    </div>
  {{/each}}
</div>
```

注意到这两个模板引用组件时的不同之处了吗？目击记录列表使用了{{yield}}为<div class="caption">添加子元素，但是神秘生物模板不需要那么做。如果只需要使用组件模板内置的元素，则可以使用**行级组件**的写法——就像上面的代码，在书写标签名{{listing-item}}时开头不加#，同样在结尾处也不需要结束标记{{/listing-item}}。

为了给组件添加链接，用{{#link-to}}包裹整个组件。在旧代码中只为图片添加了链接，而在新代码中整个组件都可以被点击，这个链接指向神秘生物详情页。这个例子展示了在不同路由模板中渲染相似内容时使用组件的灵活性。当然也可以通过给组件加入新属性，在组件中实现添加链接的功能。

25.3　数据向下，动作向上

接下来创建一个新组件，它需要跟随应用的状态变化而变化。这个组件是一个提示框，会在添加列表条目后显示一条提示信息。

组件有一个非常重要的原则，就是“数据（或状态）向下，动作向上”。组件与控制器不同，它不能直接修改应用的状态，所以它需要以动作的形式向上传递变化。而另一方面，组件从父模板接受状态数据，也就是数据向下传递（如图25-2所示）。

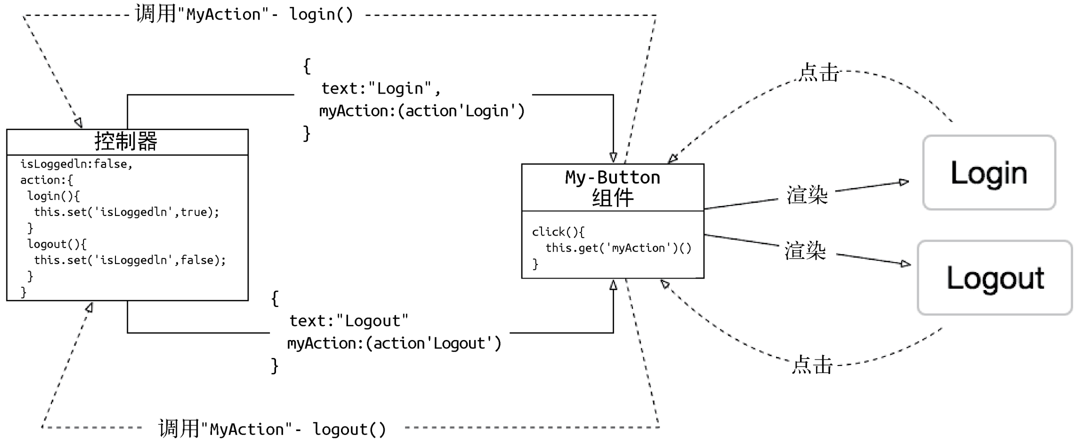


图25-2　数据向下，动作向上

当组件需要更换控制器时，可以直接把路由模型传给组件，而不需要使用控制器的装饰器或者动作。

在app/templates/application.hbs中创建新的组件和动作。在目击记录创建完成时，它们会弹出一条全局的提示消息。

首先，在终端中生成新组件：

```
ember g component flash-alert
```

`{{flash-alert}}`组件将作为一个容器，其中包含了提示信息的内容和使用`<strong>`修饰的标题。

用下面的代码替换app/templates/components/flash-alert.hbs的内容：

```
{{yield}}
<strong>{{typeTitle}}!</strong> {{message}}
```

修改app/components/flash-alert.js，为组件添加`classNames`属性：

```
import Ember from 'ember';

export default Ember.Component.extend({
  classNames: ["alert"]
```

```
});
```

修改完成后，组件会渲染出如下代码：

```
<div class="alert">
  <strong>{{typeTitle}}!</strong> {{message}}
</div>
```

25.4　类名绑定

虽然组件能正常显示信息，但是却无法根据消息的类型调整组件的样式。Bootstrap提供了许多提示框的样式和样式变体："alert-success"、"alert-info"、"alert-warning"和"alert-danger"。

为了使用这些样式，需要使用计算属性来生成类型名，使消息类型与样式类名一一对应。

首先在app/components/flash-alert.js中添加一个计算属性：

```
...
export default Ember.Component.extend({
  classNames: ["alert"],
  typeClass: Ember.computed('alertType', function() {
    return "alert-" + this.get('alertType');
  })
});
```

这里创建了一个计算属性typeClass，用于为组件的<div>容器提供类名。这个计算属性会获取组件的alertType属性（稍后添加），然后为"alertType"添加"alert-"前缀并返回，这样在使用组件时可以只传入"success"、"info"、"warning"或者"danger"等提示类型就可以获得对应样式的提示框。除了typeClass之外，后面还有一个计算属性也要使用alertType属性。

最后，修改app/components/flash-alert.js，把组件的类名与属性相互绑定：

```
...
export default Ember.Component.extend({
  classNames: ["alert"],
  classNameBindings: ['typeClass'],
  typeClass: Ember.computed('alertType', function() {
    return "alert-" + this.get('alertType');
  })
});
```

这段代码的功能是把计算属性typeClass的值作为一个类名加入到classNames数组中。这样每当alertType属性改变时，组件的样式就会相应地改变。

classNameBindings是classNames属性专用的。除此之外Ember还有一个组件属性与classNameBindings互补：attributeBindings，它可以对其他属性进行绑定。可以用这个方法把组件元素的属性与组件属性相绑定。举个列子，将href与一个计算属性绑定：

```
export default Ember.Component.extend({
  attributeBindings: ['href', 'customHREF:href'],
  href: "http://www.mydomain.com",
```

25

```
  customHREF: "http://www.mydomain.com"
});
```

通过使用属性绑定，可以把传入组件的状态数据与组件的任意属性相互绑定，更多细节可以参考Ember的文档。

接下来添加一个计算属性，把消息类型以字符串的形式渲染到页面中。由于在模板中用到了`typeTitle`，所以要在app/components/flash-alert.js中添加这个计算属性：

```
import Ember from 'ember';

export default Ember.Component.extend({
  ...
  typeClass: Ember.computed('alertType', function() {
    return "alert-" + this.get('alertType');
  }),
  typeTitle: Ember.computed('alertType', function() {
    return Ember.String.capitalize(this.get('alertType'));
  })
});
```

现在已经为提示框加入了标题，标题的内容是提示类型的大写形式，并且在标题外使用了`<strong>`标签。同时，为组件添加了一个类名和一个装饰器，它们通过计算属性对`alertType`进行处理后得到各自需要的结果。

接下来，修改app/templates/application.hbs，把组件应用到页面中：

```
<header>
  ...
</header>
<div class="container">
  {{flash-alert}}
  {{outlet}}
</div>
```

这个组件是一个行级组件，也就意味着组件不用为`{{yield}}`渲染任何代码。换句话说，根本不需要在这个组件的模板中添加`{{yield}}`标签。

25.5　数据向下

截至目前，提示框还只是一个空的容器，因为并没有给它传任何数据。打开app/templates/application.hbs，添加以下代码：

```
...
  {{flash-alert message="This is the Alert Message" alertType="success"}}
  {{outlet}}
</div>
```

启动服务器然后访问http://localhost:4200/sightings，查看`{{flash-alert}}`组件渲染后的效果（如图25-3所示）。

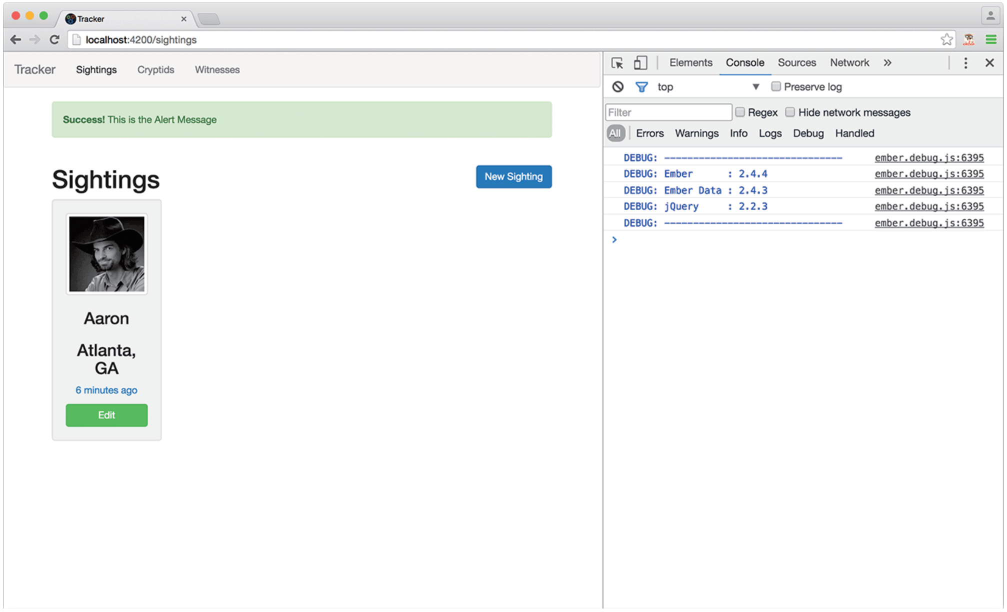


图25-3 提示信息

现在提示框已经可以在页面中显示了，下一步要把提示的内容和提示框的类型换成动态数据。首先生成一个新控制器，在终端中执行：

```
ember g controller application
```

`{{flash-alert}}`组件的状态由应用的总控制器负责管理，所以需要为控制器添加一些属性。请打开app/controllers/application.js，添加如下属性：

```
import Ember from 'ember';

export default Ember.Controller.extend({
  alertMessage: null,
  alertType: null,
  isAlertShowing: false
});
```

添加的这些属性将会通过动作进行修改。修改app/templates/application.hbs，将这些属性值传给提示框组件：

```
...
</header>
<div class="container">
  {{#if isAlertShowing}}
    {{flash-alert message="This is the Alert Message" alertType="success"}}
                          alertMessage alertType=alertType}}
  {{/if}}
  {{outlet}}
</div>
```

现在控制器的属性值可以根据动作进行修改了。只有在`isAlertShowing`为`true`且其他属性都有值时，应用才会渲染提示框。设置属性的动作是由控制器发出的，它会在应用中向上层层传递。如何才能向上传递呢？好在Ember已经内置这个机制了（如图25-4所示）。

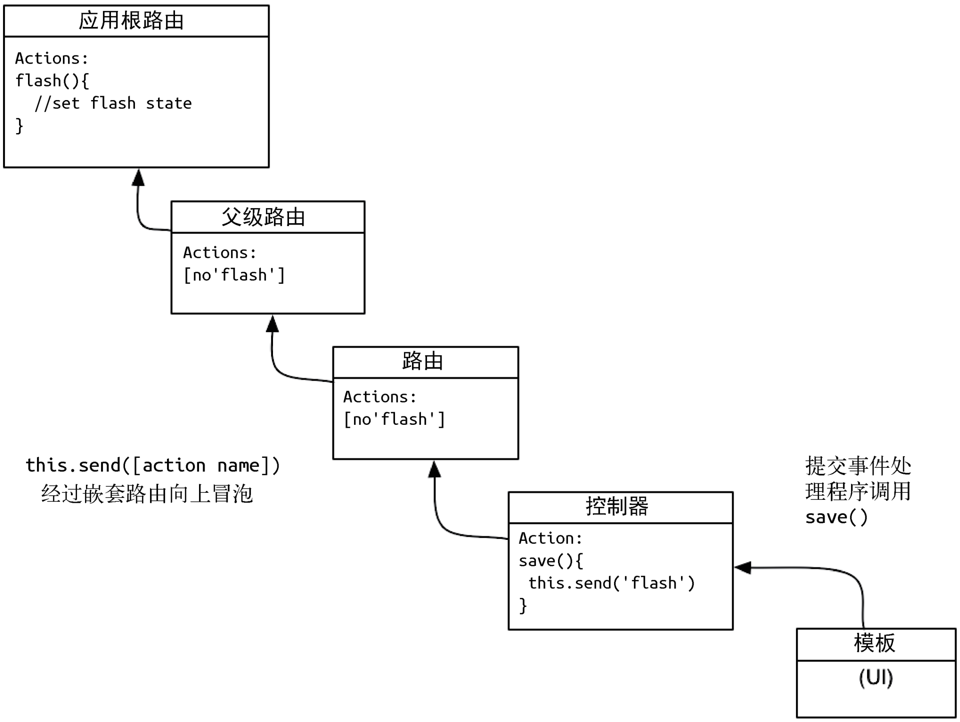


图25-4　提示信息渲染流程

需要给路由添加一个动作。在控制器中调用一个动作，然后动作会依次触发当前控制器、当前路由、父级路由和应用根路由。

由于需要在app/templates/application.hbs中使用提示框组，所以要创建一个应用根路由：

```
ember g route application
```

运行这条命令后，会看到如下提示：

```
[?] Overwrite app/templates/application.hbs?
```

请输入`n`或者`no`，否则刚创建的模板文件会被覆盖，因为只需要生成app/routes/application.js文件。

接下来对app/routes/application.js做一些修改：

```
import Ember from 'ember';

export default Ember.Route.extend({
```

```
    actions: {
      flash(data){
        this.controller.set('alertMessage', data.message);
        this.controller.set('alertType', data.alertType);
        this.controller.set('isAlertShowing', true);
      }
   }
  });
```

25.6　动作向上

前面已经为项目添加了显示提示框所用的动作，现在只需要触发这个动作并传入数据就可以在页面上展示相应的提示信息了。这里的数据是一个对象，其中包含alertType和message属性。

alertType属性的值不仅需要在页面中显示，也被用来控制提示框的样式，它可以在以下选项中取值："success"、"warning"、"info"和"danger"。从控制器中触发动作的方法如下：

```
this.send('flash', {alertType: "success", message: "You Did It! Hooray!"});
```

在用户成功添加新的目击记录后调用这个提示框。打开app/routes/sightings/new.js，添加以下代码：

```
...
    create() {
      var self = this;
      this.get('sighting').save().then(function(data){
        self.send('flash', {alertType: "success", message: "New sighting."});
        self.transitionTo('sightings');
      });
...
```

现在点击页面上的New Sighting按钮，进入创建记录的页面。在列表中任选一种神秘生物和一位目击者，输入地点，最后点击Save。当这条记录插入数据库时，应用会跳转到目击记录列表页，同时在页面的最顶部会出现一条提示信息（如图25-5所示）。

最后一步是添加动作和事件来移除消息。由于只需要在创建记录后展示提示框，所以需要在移除消息时隐藏相关元素。

在app/controllers/application.js中添加一个removeAlert动作：

```
...
export default Ember.Controller.extend({

  alertMessage: null,
  alertType: null,
  isAlertShowing: false,
  actions: {
    removeAlert(){
      this.set('alertMessage', "");
      this.set('alertType', "success");
```

```
      this.set('isAlertShowing', false);
    }
  }
});
```

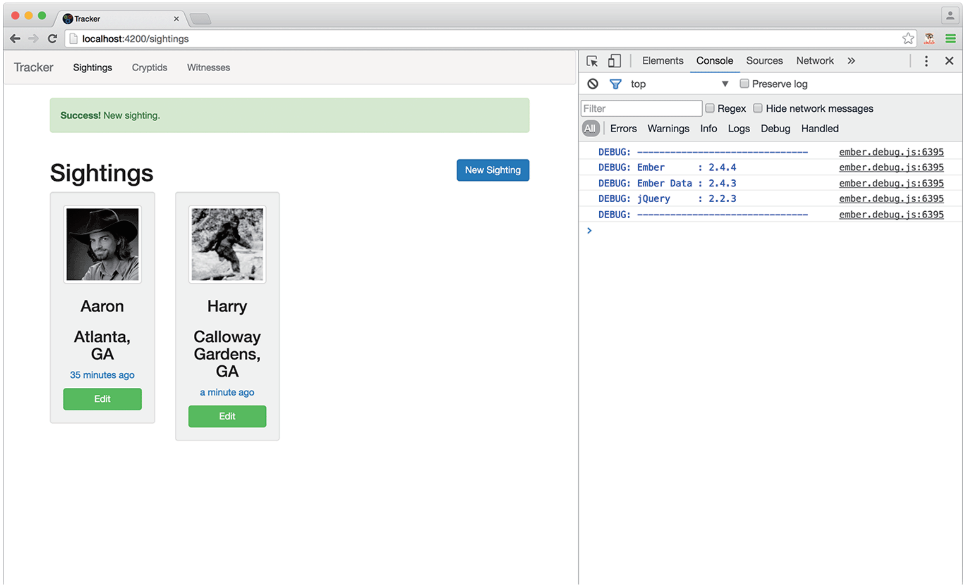


图25-5　提示信息：新目击记录

这段代码将`isAlertShowing`设置为`false`，将`alertMessage`设置为空字符串，还将`alertType`设置为`"success"`。

接下来，向组件发送`removeAlert`动作。在app/templates/application.hbs中加入以下代码：

```
...
</header>
<div class="container">
  {{#if isAlertShowing}}
    {{flash-alert message=alertMessage alertType=alertType}}
      close=(action "removeAlert")}}
  {{/if}}
  {{outlet}}
</div>
```

这里`close=(action "removeAlert")`的语法似乎有些奇怪。其实这是Ember 2.0中加入的新语法，叫作**闭包动作**（closure action）。它的功能有些类似于起别名，将函数字面量传入组件的`close`属性，然后可以在组件中直接触发动作。

如果要在旧版本的Ember中实现同样的逻辑会非常麻烦。闭包动作不是简单地将函数作为参数传递。如果想了解更多细节，可以参考EmberJS博客中一篇介绍Ember 1.13版和2.0版新特性的文章：emberjs.com/blog/2015/06/12/ember-1-13-0-released.html。

接下来在组件中触发这个动作。由于组件本身是DOM元素的实例，所以在组件中可以以键值对的形式为DOM元素定义事件。我们在组件中定义`click`方法，Ember会自动为`<div>`容器添加事件监听器，当发生点击时调用该方法。编辑app/components/flash-alert.js，加入以下代码：

```
import Ember from 'ember';

export default Ember.Component.extend({
  ...
  typeTitle: Ember.computed('alertType', function() {
    return Ember.String.capitalize(this.get('alertType'));
  }),
  click() {
    this.get('close')();
  }
});
```

当执行组件的`close`属性时，会调用控制器中的`removeAlert`方法。通过闭包动作，我们使用一个在父控制器中定义的函数为组件属性赋值，并把组件的功能绑定到了外层作用域中。你可以将提示框组件添加到任意层级的页面中，并且可以根据上下文环境为`close`动作赋予不同的功能。

现在我们已经完成了一个组件的开发，并通过它展示了组件的运作方式：数据向下，动作向上。数据向下传递以便定制组件的内容和效果，动作向上传递以便用外层控制器控制组件的状态。请在开发组件时牢记这个模式。

学完第四部分的所有内容，你便了解了什么是现代化Ember应用的架构。你还知道了MVC模式以及框架如何使用预置的对象拆分应用的逻辑。此外，你还学会了如何使用Ember提供的脚手架和构建工具，熟悉了Ember的命名模式和开发惯例。相信从现在开始，你再输入`ember new`创建新应用时肯定会更加自信，可以游刃有余地创建模型、编辑路由或是开发组件。

Ember社区共同维护着这个出色的框架，而且随着JavaScript的成长也在不断提升其效率。创造这个框架的人当初所遇到的挑战，你在不断磨炼自己JavaScript技能的过程中也会遇到。别忘了遇到麻烦就提问，可能的话帮忙修修bug，尽可能地多做一些回馈。你现在已经是火热的JavaScript社区的一员了。

25.7　初级挑战：自定义提示信息

新建目击记录后，出现的消息框`{{flash-alert}}`的内容太通用了。请在消息内容中加入新添加的目击记录发生的地点和时间。

25.8　中级挑战：将导航条转化为组件

25

请将应用的导航条（`NavBar`）也转化成一个组件，在其中包含一个属性，通过这个属性控制导航条在两种形式间切换。另外为组件添加条件语句，以便在导航条中展示自定义链接。

25.9　高级挑战：提示框数组

请将提示框组件的参数改为数组，其中可以包含多条不同类型和内容的消息，以满足同时展示多个提示框的场景的需求。在设置消息时使用`Ember.ArrayProxy`而不用再使用单独的属性。请将消息内容、消息类型、消息索引添加到数组中。（消息索引是一个新的属性，添加这个属性的目的是在用户点击消息时将其从数组中移除。）

第 26 章

后　　记

恭喜你！你马上就要读完本书了。不是所有人都像你一样自律，能够坚持学习并完成书中的项目。给自己点个赞吧！

努力终有回报，你已经是一个前端开发者了。

26.1　最后的挑战

还有最后一个挑战：成为一个**优秀的**前端开发者。优秀的开发者都各有其优秀的方式，所以你也要找到自己的方式。

那么从何开始呢？下面有一些建议。

写代码。如果你不运用前面所学的知识，你很快就会忘记它们。为一个项目贡献代码，或自己写一个简单应用。不管做什么，不要浪费时间：写代码。

学习。你已经掌握了书中许多知识的一小部分，它们是否激发了你的灵感呢？运用你最喜欢的技术写一些代码；查找、阅读相关文档——如果没有文档，那就去读读书。另外，去JavaScript Jabber播客节目（devchat.tv/js-jabber），找一些最近和前端开发相关的有趣又有料的讨论。

与人交流。参加本地的交流会，在这种场合能遇到兴趣相投的开发者。另外，很多高级前端开发者在Twitter上都很活跃。还可以参加前端会议，遇见更多开发者。（也许会见到我们哦！）

探索开源社区。前端开发在www.github.com上正蓬勃发展着。如果你发现一个很酷的库，不妨看看它的贡献者参与的其他项目。你也要积极分享自己的代码——说不定就有人会觉得你写的代码很好用或者很有趣。通过WDRL（Web Development Reading List，Web开发阅读清单）邮件列表，你还可以了解到前端社区（wdrl.info）正在发生的事情。

26.2　插播一个广告

你可以在Twitter上找到我们。Chris是@radishmouse，Todd是@tgandee。

如果你喜欢这本书，可以去www.bignerdranch.com/books看看Big Nerd Ranch Guides系列的其他图书。我们还为开发者提供了一系列的一周课程，只用一周就能轻松学习一本书的精华部分。当然，假如你只需要找人开发很棒的代码，我们也提供外包编程服务。更多信息请访问

www.bignerdranch.com。

26.3　感谢你

如果没有像你一样的读者，就没有我们这本书。谢谢购买并阅读了本书。

专业 —— 源自大名鼎鼎的Big Nerd Ranch训练营实战课程，该训练营已经为微软、Google、Facebook等行业巨头培养了众多专业人才。

领先 —— 涵盖前端开发先进的技术，实现精彩Web应用。

实战 —— 4大Web开发实战项目，以项目驱动讲解，以实践引领理论。

梯度 —— 从基础的交互式网页到实时聊天应用，由浅入深，横跨大前端。

“本书将时下流行的技术融入4个实战项目中，深入浅出地讲解了Web开发的整个过程，既重视理论，又切合实战。通过学习，你不仅可以更好地理解JavaScript、CSS3和HTML5，更能创造出真正跨平台的Web应用，将胜利的果实呈现在用户面前。总之，不管你之前有没有Web开发经验，本书都能让你受益匪浅。”

—— 田爱娜，HTML5梦工场、iWeb学院创始人

“我非常推崇这本书的写作风格和它明了易懂的示例，作者通过逐步提高项目的难度引导读者学习。”

—— Amazon读者

“这是我的第3本Big Nerd Ranch Guide系列图书，不得不说，他们的讲解方法真是太棒了！我在阅读的过程中就能运用学到的大部分内容，而且项目的展开方式、作者使用的命名约定、文件夹结构和代码组织也都是亮点！”

—— Amazon读者

Chris Aquino

Web开发专家，Big Nerd Ranch讲师。作为开发者，他希望给用户提供有意义的数据体验；作为讲师，他致力于帮助他的团队和学生构建出更好的Web。平时喜爱发条玩具、浓缩咖啡和各式烧烤。

Todd Gandee

前端工程师，Big Nerd Ranch讲师。拥有十余年Web顾问经验，专业技能娴熟。业余时间喜欢跑步、骑行以及攀岩。

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议	计算机 / 软件开发 / Web开发

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-46616-7

ISBN 978-7-115-46616-7
定价：99.00元